# SIMULACIÓN Y EVALUACIÓN DEL DESEMPEÑO DE DOS VARIANTES DE TCP-POR DEFECTO Y TCP-LINUX EN NS-2

**LUZ KARIME RAMÍREZ ARAGÓN**

**UNIVERSIDAD DEL VALLE**
**FACULTAD DE INGENIERÍA**
**ESCUELA DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA**
**PROGRAMA ACADÉMICO DE INGENIERÍA ELECTRÓNICA**
**SANTIAGO DE CALI**
**2014**

# SIMULACIÓN Y EVALUACIÓN DEL DESEMPEÑO DE DOS VARIANTES DE TCP-POR DEFECTO Y TCP-LINUX EN NS-2

**LUZ KARIME RAMÍREZ ARAGÓN**

**Trabajo de grado para obtener el título de:**
**Ingeniera Electrónica**

**Director**
**Ing. Oscar Polanco Sarmiento**

**UNIVERSIDAD DEL VALLE**
**FACULTAD DE INGENIERÍA**
**ESCUELA DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA**
**PROGRAMA ACADÉMICO DE INGENIERÍA ELECTRÓNICA**
**SANTIAGO DE CALI**
**2014**

## NOTA DE ACEPTACIÓN

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

_____
**FIRMA DEL DIRECTOR**

_____
**FIRMA DEL JURADO**

_____
**FIRMA DEL JURADO**

**A mis padres por su amor y apoyo incondicional.**

**A mis hermanos por su colaboración.**

**A mi novio por su paciencia y apoyo en este proceso.**

**A mi director de trabajo de grado por esta segunda oportunidad.**

## AGRADECIMIENTOS

A mi madre Fabiola Aragón Correa y a mi padre José Rubidio Ramírez Acevedo quienes con mucho esfuerzo, dedicación y amor me han llevado por un buen camino en compañía de mis hermanos Diego Fernando y Maribel quienes siempre me han brindado un amor incondicional. En especial gracias a mi hermana Maribel quien permitió que siguiera mis estudios universitarios a pesar de las circunstancias y a mi cuñado Jhon Edwin Tobón quien nos ha brindado un apoyo desde el momento que hizo parte de nuestra familia.

A Hernando Moreno Padilla, que más que mi novio ha sido mi compañero, mi amigo, mi confidente. A él gracias por su paciencia, por soportarme durante esta dura etapa final de mi carrera profesional y por estar a mi lado brindándome su apoyo y confianza.

A mi familia quienes siempre han estado presentes en las etapas de mi vida y quienes confían en mi capacidad de superar barreras y de crecer profesionalmente.

A mi director de tesis, el profesor Oscar Polanco, quien me escuchó y decidió brindarme una segunda oportunidad para poder culminar mi carrera. A él gracias por el tiempo de tutoría y por sus palabras en el momento indicado.

A mis amigos y compañeros quienes nunca dudaron de mi capacidad para lograr culminar mi trabajo de grado, quienes estuvieron en mis malos ratos y nunca dejaron de brindarme su apoyo: Iván Hernando González, Iván Darío Montoya, Henry Eduardo Sarria, Juan Manuel Arbeláez, Johannes Allen Sánchez, Darío Narváez, Stephen Ramírez, a nuestro gran amigo y compañero Jonathan Bohórquez Alzate (QEPD).

A mis amigos y amigas quienes solo han tenido palabras de aliento para sacar adelante mi carrera que en algún momento di como perdida.

A Dios por permitirme compartir mi tiempo y mi vida con personas tan maravillosas.

## RESUMEN

Actualmente, en los procesos de investigación, enseñanza y aprendizaje en el campo de las comunicaciones, el uso de las herramientas de simulación son de gran importancia, debido a que representan una opción económica para experimentar y evaluar el desempeño de una red, sin necesidad de contar con una infraestructura para su implementación. Facilitando el estudio de las redes de comunicación mediante el desarrollo, diseño y análisis de redes.

Este documento presenta un escenario en el que se modela una topología de red de datos con tráfico TCP, sometida a condiciones de congestión mediante un modelo de error en el enlace cuello de botella, para lo cual se usa el lenguaje de programación OTcl (versión orientada a objetos de Tcl (*Tool Command Language*)) que posteriormente es simulado en el software de simulación Network Simulator 2 (NS-2)

En la topología de red se incorporan las variantes Reno y New Reno del protocolo TCP (*Transmission Control Protocol*) de dos formas diferentes:

1. Usando la implementación TCP propia del simulador NS-2, que denominaremos variante TCP por defecto en NS-2.
2. Usando la implementación del módulo TCP-Linux incorporada en NS-2, que denominaremos variante TCP-Linux.

Finalmente se evalúa el desempeño de la red para cada variante (Reno y New Reno) en sus dos implementaciones (TCP por defecto y TCP-Linux en NS-2) mediante la observación y el análisis de dos métricas: Ventana de Congestión (*cwnd*) y Rendimiento (*Throughput*).

## TABLA DE CONTENIDO

## LISTA DE FIGURAS

## LISTA DE TABLAS

# 1. INTRODUCCIÓN

Network Simulator 2 (NS-2) se ha convertido en una de las herramientas de simulación más utilizada en el ámbito académico y de investigación gracias a que se ha distribuido como software libre y de código abierto, lo que ha permitido que estudiantes, ingenieros e investigadores aporten scripts que permiten que este siempre en proceso de desarrollo [1].

NS-2 permite simular un amplio número de protocolos para redes de telecomunicaciones y de datos, entre ellos el protocolo TCP. Implementa las diferentes versiones del protocolo TCP que se han propuesto para mejorar la respuesta del algoritmo de control de congestión y de la transmisión en general, por tanto es posible implementar diseños de red que permitan cuantificar el desempeño del protocolo TCP bajo condiciones controladas.

La simulación y evaluación del desempeño de diferentes variantes del protocolo TCP en NS-2 en una red bajo diversas condiciones y el entendimiento del efecto que tienen los diferentes factores sobre el desempeño de la misma, permitirá abordar otros temas en un futuro y abre la posibilidad de diseñar y desarrollar diferentes módulos de simulación donde se pueda modelar, simular y experimentar con diversos escenarios.

## 1.1 NETWORK SIMULATOR 2 (NS-2)

NS-2 es un simulador de eventos discretos (suceso que se genera en un instante de tiempo particular) orientado a objetos, escrito en C++ con un intérprete OTcl (versión orientada a objetos de Tcl: Tool Command Language).

Está basado en dos lenguajes de programación, **C++** (a nivel de datos) y **OTcl** (a nivel de control). El núcleo principal de NS-2 está escrito en C++, lo cual agiliza la simulación y el lenguaje OTcl se utiliza para actuar sobre el simulador. NS-2 está implementado en **C++**, mediante el cual los usuarios pueden crear nuevas clases, construir sus propios protocolos y modificar según sus necesidades los ya existentes. Su uso se debe a que es un lenguaje compilado más eficiente en tiempo de ejecución.

NS-2 usa OTcl para desarrollar los scripts de simulación, donde se configuran las características de los escenarios y se establecen los parámetros de la simulación. Este lenguaje interpretado permite realizar cambios de manera rápida e interactiva lo que es importante en las continuas modificaciones que se tiene que realizar en los parámetros de configuración y en la exploración de diferentes escenarios.

### 1.1.1 Elementos de Red

Los elementos básicos de una red son los nodos y enlaces que permiten definir una topología de red. Los nodos en NS-2 actúan como un host cuando se configuran como nodos fuente o destino y actúan como router cuando en la configuración son nodos intermedios [2]. Un enlace es el encargado de conectar los y transportar los paquetes entre ellos desde una fuente hasta un destino. En la Tabla 1.1 se presentan los elementos de red y su configuración en NS-2.

**Tabla 1.1 Elementos de Red en NS-2**

| Elemento | Configuración NS-2 |
|---|---|
| Nodo | `$ns` **`node`** |
| Enlace Simple | `$ns simplex-link $n1 $n2 <bandwidth> <delay> <queue_type>` |
| Enlace duplex | `$ns duplex-link $n1 $n2 <bandwidth> <delay> <queue_type>` |

En la Tabla 1.1 se especifica cómo crear un enlace en NS-2, donde **`simplex-link/duplex-link`** crea un enlace unidireccional o bidireccional conectando los nodos `n1` y `n2`. **`<bandwidth>`** determina el ancho de banda del enlace en bits por segundo [bps]. **`<delay>`** determina el retardo de propagación del enlace en segundos [s]. **`<queue_type>`** determina el tipo de cola en el enlace.

### 1.1.2 Protocolos

En la Tabla 1.2 se presentan los protocolos que están implementados en NS-2. Los detalles de estos protocolos se pueden encontrar en el manual de NS [3].

**Tabla 1.2 Protocolos en NS-2**

| Protocolo | | Agente en NS-2 |
|---|---|---|
| TCP | Transmission Control Protocol | Agent/TCP |
| UDP | User Datagram Protocol | Agent/UDP |
| SCTP | Stream Control Transmission Protocol | Agent/SCTP |
| SRM | Scalable Reliable Multicast | Agent/SRM |
| DCCP | Datagram Congestion Control Protocol | Agent/DCCP/ |
| RTP | Real-time Transport Protocol | Agent/rtProto |

En la Tabla 1.3 se presentas los agentes de TCP soportados en NS-2

**Tabla 1.3 Agentes TCP en NS-2**

| Agentes Emisores | Agentes Receptores |
|---|---|
| Agent/TCP/Reno | Agent/TCPSink |
| Agent/TCP/Newreno | |
| Agent/TCP/Sack1 | Agent/TCPSink/DelAck |
| Agent/TCP/Vegas | |
| Agent/TCP/Fack | Agent/TCPSink/Sack1 |
| Agent/TCP/Linux | Agent/TCPSink/Sack1/DelAck |

En la Tabla 1.4 se presentan algunas de las variables del Agent/TCP y Agent/TCPSink.

**Tabla 1.4 Variables en NS-2**

| Variable OTcl | Valor por defecto | Descripción |
|---|---|---|
| window_ | 20 | Límite superior del tamaño de la ventana |
| numdupacks_ | 3 | Número de ACKs duplicados el cual activa Fast Retransmit |
| packetSize_ | 1000 | Tamaño del paquete TCP en bytes |
| cwnd_ | 0 | Tamaño de la ventana de congestión en paquetes |
| ssthresh_ | 0 | Umbral de arranque lento |
| bytes_ | 0 | Número de bytes recibidos |
| ndatapack_ | 0 | Número de paquetes transmitidos por la conexión |

### 1.1.3 Aplicaciones

Son los encargados de simular el flujo de tráfico en la red. Las aplicaciones en NS-2 se clasifican en generadores de tráfico y aplicaciones simuladas. Los generadores de tráfico tienen una programación predefinida para la generación de la carga útil en la red mientras que las aplicaciones simuladas actúan como si una aplicación real se estuviera ejecutando en NS-2 [2]. En la Tabla 1.5 se presentan las aplicaciones en NS-2

**Tabla 1.5 Aplicaciones en NS-2**

| Generador de Tráfico | Otcl | Aplicación Simulada | Otcl |
|---|---|---|---|
| CBR (Constant Bit Rate) | `Application/Traffic/CBR` | FTP (File Transfer Protocol) | `Application/FTP` |
| Exponential On/Off | `Application/Traffic/Exponential` | | |
| Pareto On/Off | `Application/Traffic/Pareto` | Telnet | `Application/Telnet` |
| Traffic Trace | `Application/Traffic/Trace` | | |

### 1.1.4 Modelo de Error

Es un módulo en NS-2 utilizado para imponer error en la transmisión de paquetes mediante la pérdida de paquetes en la simulación [2]. En las simulaciones, los errores pueden ser generados por modelos simples de error tal como la tasa de error de paquetes o por modelos estadísticos [3]. En la Tabla 1.6 se presentan los modelos de error definidos en NS-2

**Tabla 1.6 Modelos de Error en NS-2**

| Modelo de Error | Descripción |
|---|---|
| `ErrorModel/Uniform` | Modelo de error Uniforme |
| `ErrorModel/Expo` | Modelo de error de dos estados; cada estado está representado por una variable aleatoria exponencial |
| `ErrorModel/Empirical` | Modelo de error de dos estados; cada estado está representado por una variable aleatoria empírica |
| `ErrorModel/TwoStateMarkov` | Modelo de error de dos estados; donde el tiempo de permanencia del estado es exponencial |

### 1.1.5 Archivos

Los archivos generados por el simulador contienen grandes datos de los eventos que ocurren durante la simulación de la red. La traza del archivo de salida de NS-2 (out.tr) se muestra en la Figura 1.1

| (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) | (10) | (11) | (12) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Evento | Tiempo | Nodo entrada | Nodo salida | Tipo Pkt | Tamaño Pkt | Bandera | ID flujo | Dir. Origen | Dir. Destino | Numero de Secuencia | ID Pkt |

**Figura 1.1 Estructura archivo .tr en NS-2**

A continuación se describen cada uno de los campos de la traza del archivo out.tr

**(1)** Evento que el paquete experimenta en la simulación: **"+"** (en cola), **"-"** (decolado), **"r"** (recibido), **"d"** (descartado).

**(2)** Tiempo en el que ocurre el evento.

**(3)** y **(4)** Nodos de entrada y salida de la conexión en la que se produce el evento.

**(5)** y **(6)** Tipo y tamaño del paquete (pkt). El tipo puede ser: tcp,ack,cbr

**(7)** Bandera que indican un comportamiento diferente en la red. Ejemplo: **"A"** que indica acción de congestión en la red.

**(8)** Identificador del flujo. El usuario lo puede establecer en el script de configuración.

**(9)** y **(10)** Indican las direcciones del nodo origen y destino del paquete.

**(11)** Numero de secuencia del paquete.

**(12)** Identificador único del paquete. NS-2 asigna un nuevo identificador para cada nuevo paquete creado en la simulación.

Para el análisis de los resultados de la simulación se debe filtrar la información de los archivos de salida, con el propósito de extraer subconjuntos de datos de interés. Mediante el uso de los lenguajes PERL y AWK se puede realizar los filtros para observar y analizar las métricas de interés.

## 1.2 TCP (Transmission Control Protocol)

El Protocolo de Control de Transmisión (TCP) es uno de los principales protocolos de la capa de transporte del modelo TCP/IP. Es el protocolo de transferencia fiable de datos, orientado a la conexión de la capa de transporte de Internet [4].

Es orientado a la conexión porque TCP necesita una fase de establecimiento de la conexión explícita durante la cual los dos lados de la conexión están de acuerdo para intercambiar datos entre sí. También tiene una fase de desmontaje de la conexión requerida para que cada host sepa que puede liberar ese estado [5].

Durante la conexión se envían segmentos para establecer parámetros que aseguren la transferencia de datos y ambos lados de la conexión inician variables de estado TCP asociadas con la conexión TCP [4].

TCP garantiza la entrega fiable y en orden de un flujo de bytes. Es un protocolo *full duplex*, lo que significa que cada conexión TCP soporta un par de flujos de bytes, uno fluyendo en cada dirección. Por lo tanto, cada extremo de una conexión debe mantener un número de secuencia de los datos que fluyen en cada dirección [6].

TCP implementa un mecanismo de control de flujo que permite al receptor limitar la cantidad de datos que el emisor puede transmitir para evitar que el emisor sature la capacidad del buffer del receptor. También implementa un mecanismo de control de congestión para regular qué tan rápido se envían los datos y evitar que el emisor inyecte demasiado tráfico en la red.

### 1.2.1 Control de Flujo

TCP implementa control de flujo basándose en el campo tamaño de ventana anunciada proporcionada por un receptor en cada ACK que envía al emisor. Mediante la ventana anunciada se indica cuánto espacio de bufer queda en el receptor. De esta manera el emisor adapta su tasa de envío para evitar el rebasamiento en el buffer del receptor [6].

### 1.2.2 Control de Congestión

La congestión de la red se presenta cuando hay demasiados paquetes almacenados en los buffers de los nodos intermedios esperando ser transmitidos. De esta manera los routers se ven obligados a descartar los paquetes y el rendimiento de la red disminuye. En TCP la pérdida de paquetes es una indicación de congestión.

TCP implementa un conjunto de algoritmos para determinar cuándo se debe reducir la velocidad, con el fin de evitar que la red se sature. El control de congestión en TCP se basa en el tamaño de la ventana de congestión (*cwnd*), utilizada por el emisor o la fuente para limitar la cantidad de datos que se le permite tener en tránsito en un momento dado [5].

Las implementaciones de TCP actuales incluyen cuatro algoritmos: arranque lento (*Slow Start*), prevención de congestión (*Congestion Avoidance*), retransmisión rápida (*Fast Retransmit*) y recuperación rápida (*Fast Recovery*).

### 1.2.3 Arranque lento (*Slow Start*)

Al inicio de la conexión, el emisor desconoce el estado de la red. El algoritmo *Slow Start* pone a prueba la capacidad de la red mediante el aumento progresivo del ritmo de envío de paquetes hasta detectar la congestión.

El algoritmo de *Slow Start* se ejecuta cuando se crea una nueva conexión TCP o cuando se ha detectado pérdida debido a un tiempo de retransmisión (*timeout*). Cuando se inicia la conexión TCP, la ventana de congestión tiene un valor de un paquete (*cwnd*=1), la fase de *Slow Start* incrementa este valor en un segmento por cada ACK recibido, es decir que duplica *cwnd* cada RTT, por tanto *cwnd* crece exponencialmente. La fase de *Slow Start* termina cuando se detecta una pérdida (por *timeout* o ACKs duplicados), en este momento *cwnd* se reduce a la mitad y se establece el valor del umbral de arranque lento (*ssthresh*).

En caso de que la detección de la pérdida del paquete sea por *timeout*, se reestablece *cwnd*=1 y se inicia de nuevo la fase de *Slow Start* que llegará hasta el valor de *ssthresh*. Si la pérdida es por ACKs duplicados, el valor de *cwnd* se reduce a la mitad y se inicia el algoritmo *Congestion Avoidance*.

### 1.2.4 Prevención de Congestión (*Congestion Avoidance*)

Una vez se conoce la capacidad de la red, se trata de evitar congestión con el algoritmo *Congestion Avoidance*.

Cuando *cwnd* este en el nivel del umbral (*cwnd=ssthresh*), TCP ejecuta el algoritmo de *Congestion Avoidance* que busca la capacidad adicional mediante el incremento de aproximadamente un segmento por cada ventana reconocida, por tanto *cwnd* crece levemente con nuevas llegadas de ACKs. Por tanto el crecimiento es aproximadamente lineal.

Normalmente una conexión TCP siempre está ejecutando uno de estos dos algoritmos. El punto de conmutación entre el algoritmo *Slow Start* y *Congestion Avoidance* es el umbral de arranque lento (*ssthresh*). El principal objetivo de *ssthresh* es recordar la última "mejor" estimación de la ventana de operación cuando no había pérdida.

El umbral de arranque lento es un límite sobre el valor de *cwnd* que determina el algoritmo que está en funcionamiento.

Si $cwnd < ssthresh$ entonces se ejecuta el algoritmo *Slow Start*.

Si $cwnd \geq ssthresh$ entonces se ejecuta el algoritmo *Congestion Avoidance*.

Estos algoritmos están basados en los principios de conservación de paquetes y sincronización de ACKs.

### 1.2.5 Retransmisión Rápida (*Fast Retransmit*)

La retransmisión de paquetes perdidos se puede realizar de una forma más rápida y eficiente mediante el algoritmo de *Fast Retransmit*. Con este mecanismo no es necesario esperar a que venza el temporizador de retransmisión (*timeout)* para retransmitir un paquete perdido.

La retransmisión se realiza cuando el emisor (fuente) recibe un pequeño número de ACKs duplicados denominado umbral de ACKs duplicados o *DupThresh*. Normalmente este valor es constante (*DupThresh*=3), sin embargo algunas implementaciones no estándar, incluyendo Linux, altera este valor basado en el reordenamiento [6].

### 1.2.6 Recuperación Rápida *(Fast Recovery)*

Cuando se detecta congestión por el mecanismo de *Fast Retransmit* no se reestablece la ventana de congestión a un paquete ya que el algoritmo de *Fast Recovery*, permite utilizar los ACKs que todavía están en la tubería para controlar la transmisión de nuevos paquetes hasta que llegue un ACK no duplicado (ACK nuevo). De esta manera la fase de *Slow Start* se elimina eficazmente, iniciando el algoritmo de *Congestion Avoidance*. Normalmente *Fast Retransmit* se implementa conjuntamente con ***Fast** Recovery**.***

Al entrar en la fase *Fast Retransmit* / *Fast Recovery* se realizan las siguientes acciones:

1. Se asigna a *ssthresh* la mitad del valor de la ventana de congestión, *cwnd*.

2. Se retransmite el paquete perdido (*Fast Retransmit*) y se establece la ventana de congestión: $cwnd = ssthresh + 3 * SMSS$.
   Esto infla la ventana de congestión por el número de segmentos que han salido de la red.

3. Cuando se recibe otro ACK duplicado, *cwnd* se incrementa en un segmento. Esto infla artificialmente la ventana de congestión para tener en cuenta el hecho de que la recepción de cada ACK duplicado indica que un paquete ha salido de la red y si la ventana lo permite se transmite un nuevo paquete.

4. Cuando llega el primer ACK no duplicado (ACK nuevo) el valor de la ventana de congestión se establece en el umbral de arranque lento (*cwnd*=*ssthresh*) y continua con un aumento lineal de *cwnd* (*Congestion Avoidance*). Al recibir un ACK nuevo se reconoce el paquete perdido y termina la fase de *Fast Recovery*.

En la Figura 1.2 se presentan las fases de los cuatro algoritmos descritos mediante la implementación de la topología de red con un agente TCP Reno.

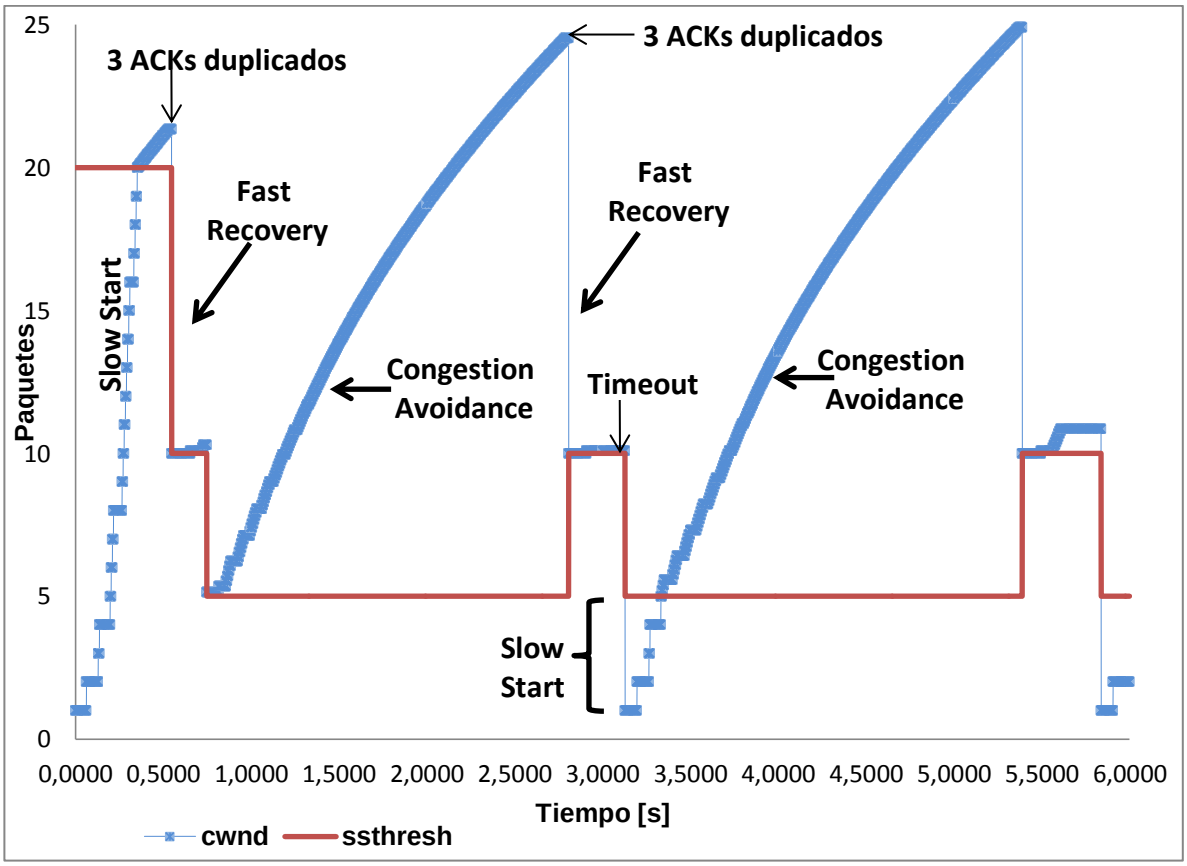


**Figura 1.2 Control de Congestión de TCP**

## 1.3 Variantes de TCP

El protocolo TCP desde sus inicios ha sufrido diferentes cambios, con el fin de mejorarlo. Distintas versiones e implementaciones de TCP se han propuesto para mejorar la función de respuesta del algoritmo de control de congestión AIMD (Additive Increase / Multiplicative Decrease), aportando algunas modificaciones significativas que mejoran aspectos de la transmisión en general. En la Tabla 1.7 se presentan las variantes TCP implementadas actualmente.

La segunda columna indica si el algoritmo de control de congestión considera la pérdida de paquete o un retardo como una indicación de la congestión. La tercera columna indica para que entornos de red se han optimizado los algoritmos de

control de congestión en la variante [7]. La cuarta columna indica si la variante esta implementada en el módulo Linux TCP.

**Tabla 1.7 Variantes TCP**

| Variante | Indicación de congestión por | Red | Implementación Linux |
|---|---|---|---|
| Tahoe | Perdida | Cableada | No aplica |
| Reno | Perdida | Cableada | ✓ |
| New Reno | Perdida | Cableada | ✓ |
| Vegas | Retardo | Cableada | ✓ |
| BIC | Perdida | Altos BDP | ✓ |
| CUBIC | Perdida | Altos BDP | ✓ |
| HSTCP | Perdida | Altos BDP | ✓ |
| HTCP | Retardo | Altos BDP | ✓ |
| HYBLA | Retardo | Altos BDP | ✓ |
| Scalable | Perdida | Altos BDP | ✓ |
| Westwood | Retardo | Inalámbrica | ✓ |
| Veno | Retardo | Inalámbrica | ✓ |
| LP | Retardo | No aplica | ✓ |
| YeAH | Retardo | Altos BDP | ✓ |
| Illinois | Perdida - Retardo | Altos BDP | ✓ |

A continuación se describen brevemente las variantes implementadas en la simulación.

### 1.3.1 TCP Reno

Esta versión mejora en cuanto a implementaciones anteriores de TCP que implementan los mecanismos AIMD (Old-Tahoe) y los algoritmos de *Slow Start*, *Congestion Avoidance* y *Fast Retransmit* (Tahoe).

Reno implementa en su algoritmo de control de congestión el mecanismo *Fast Recovery* que evita que tras una retransmisión se inicie la fase de *Slow Start*. Durante la fase de *Fast Recovery*, TCP retransmite el paquete perdido que fue detectado por 3 ACKs duplicados (*Fast Retransmit*). Cuando el emisor recibe un ACK nuevo (ACK parcial) sale de la fase *Fast Recovery*, ya que el paquete perdido que provocó la retransmisión es reconocido.

El algoritmo de *Fast Recovery* de Reno está optimizado para la pérdida de un solo paquete en una ventana de datos. El emisor transmite un paquete perdido por RTT.

### 1.3.2 TCP New Reno

La variante New Reno mejora el algoritmo de Reno mediante la introducción de una variable de recuperación en el mecanismo *Fast Recovery*. Esta variable denominada Punto de Recuperación, registra el paquete con el número de secuencia más alto que ha sido enviado después de la perdida y que aún no ha sido reconocido. De esta manera cuando el emisor recibe un ACK parcial que reconoce algunos pero no todos los paquetes que estaban en circulación al inicio de la fase de *Fast Recovery*, se trata como una indicación de que existe otro paquete perdido que debe ser retransmitido.

New Reno permanece en *Fast Recovery* hasta que todos los paquetes perdidos de esa ventana se hayan retransmitido. Cuando se recibe un ACK mayor o igual al Punto de Recuperación se termina la fase de *Fast Recovery* y se inicia el algoritmo de *Congestion Avoidance*.

En New Reno se tiene:

Si $ACK < Punto\ de\ Recuperación$ entonces es un ACK parcial.

Si $ACK \geq Punto\ de\ Recuperación$ entonces es un ACK nuevo.

### 1.3.3 TCP Linux

Una implementación de Linux TCP en NS-2 (NS-2 TCP Linux) es un módulo desarrollado por el Network Laboratory, California Institute of Technology, USA [8]. En el kernel de Linux (v2.6.22.3) trece variantes Linux TCP se pueden ejecutar en NS-2 [8] [9]. NS-2 TCP Linux introduce un nuevo TCPAgent, `LinuxTCPAgent`, que implementa la interfaz del módulo de control de congestión Linux [9].

En la Tabla 1.8 se presentan las variantes implementadas en Linux TCP en NS-2 y el respectivo nombre que se debe configurar el Agente/TCP/Linux en NS-2 [10].

**Tabla 1.8 Variantes Linux TCP en NS-2**

| Variante | select_ca |
|---|---|
| Reno | naive_reno |
| New Reno | reno |
| Vegas | vegas |
| BIC | bic |
| CUBIC | cubic |
| HSTCP | highspeed |
| HTCP | htcp |
| HYBLA | hybla |
| Scalable | scalable |
| Westwood | westwood |
| Veno | veno |
| LP | lp |
| YeAH | yeah |
| Illinois | illinois |
| C-TCP | compound |

Linux-TCP combina las características de las diferentes especificaciones de IETF con ciertas características de diseño por fuera de las especificaciones, con el propósito de mejorar el rendimiento y la eficiencia de TCP [11].

A continuación se presentan algunas de las mejoras implementadas en los algoritmos de congestión de Linux-TCP.

- **Marca de tiempo (*Timestamp*)**: Esta opción permite mediciones de tiempo de ida y vuelta más precisas. El emisor TCP puede medir exactamente los RTT para los segmentos y deducir el *timeout* estimado. También permite la detección de retransmisiones innecesarias y el emisor TCP puede continuar enviando nuevos datos y revertir los cambios en los parámetros de control de congestión [8] [11].

- **Escalado de Ventana (*Window Scaling*)**: El tamaño máximo de la ventana se aumenta al extender el campo de ventana de 16 bits de la cabecera TCP a un valor de 30 bits. Esta opción permite a TCP a utilizar mejor la red con un gran producto de ancho de banda-retardo (bandwidth delay product) [8] [11].

- **SACK (*Selective ACK*)**: Reconocimiento Selectivo permite al emisor TCP realizar el control de congestión con más precisión, mediante la detección de pérdida de paquetes con la información adicional [8] [11].

- **FACK (*Forward ACK*)**: Reconocimiento Delantero tiene un enfoque más agresivo y considera los agujeros no reconocidos entre los bloques SACK como paquetes perdidos. Aunque este enfoque a menudo resulta en un mejor rendimiento de TCP que el enfoque conservador, es demasiado agresivo si los paquetes se han reordenado en la red [8] [11].

- **D-SACK (*Duplicate-SACK*)**: Duplicado-SACK se utiliza para inferir el grado de la reordenación de paquetes y se ajusta el umbral de ACK duplicado. El emisor TCP puede concluir en ciertas circunstancias si se ha reducido innecesariamente sus parámetros de control de congestión y así revertir los parámetros a los valores anteriores a la retransmisión [8] [11].

- **ECN (*Explicit Congestion Notification*)**: Notificación Explicita de Congestión permite detectar el inicio de la congestión antes de una pérdida de paquetes. *ECN* es una manera para que los routers marquen paquetes indicando el inicio de la congestión [6] [11].

- **Ventana Inicial grande (*Larger Initial Window*)**: Basado en el tamaño determinado de SMS, la ventana de congestión se establece inicialmente en un máximo de cuatro MSS. Esta extensión ayuda a abrir su ventana de TCP más rápidamente al comienzo de cada conexión [8].
- **Cuenta de bytes apropiada (*Appropriate Byte Counting: ABC*)**: La ventana de congestión se actualiza con base en el número de bytes que están cubiertos por cada ACK recibido. Esta extensión ayuda a TCP a minimizar el impacto de los ACKs perdidos en la actualización de la ventana de congestión [6] [8].

- **Reconocimiento retardado (*Delayed Acknowledgement)***: Mediante el campo ACK acumulativo, TCP no proporciona un ACK para cada paquete. El uso de un ACK acumulativo permite retrasar el envío de un ACK por cierta cantidad de tiempo, las especificaciones TCP establecen que el retardo debe ser menor a 500ms. Muchas implementaciones utilizan un máximo de 200ms. Una proporción de 2 a 1 es bastante común para las

transferencias masivas [6]. Linux utiliza un algoritmo de ajuste dinámico para retrasar ACKs mediante el cual puede cambiar entre reconocimiento de cada segmento y el modo de ACKs retardados convencional. Al inicio de la conexión Linux deshabilita el retardo de los reconocimientos con el fin de acelerar la transmisión en la fase de *Slow Start*. Esto se llama reconocimientos rápidos [11].

- **Retransmisión Rápida con SACK (*Fast Retransmit with SACK*)**: La primera pérdida de datos se detecta y se retransmite basado en bloques SACK acumulados en lugar del esquema convencional de 3 ACKs duplicados. Esta extensión ayuda a TCP a acelerar el proceso de detección de pérdida [8]

- **Recuperación rápida con SACK (*Fast Recovery with SACK*)**: Las pérdidas de otros datos se detectan y retransmiten sin tener que esperar otro RTT. Esta extensión ayuda a TCP a acelerar el proceso de recuperación de pérdida. [10]

- **Recuperación rápida con FACK (*Fast Recovery with FACK*)**: Después de retransmisión rápida Linux implementa recuperación rápida disminuyendo el tamaño de la ventana de congestión gradualmente por un segmento y permite que un segmento sea transmitido por cada dos ACKs duplicados recibidos (Rate Halving). Esto causa que la recuperación TCP haya enviado la cantidad apropiada de datos al final de la fase de recuperación de una manera uniforme.

# 2. OBJETIVOS

## 2.1 Objetivo General:

Desarrollar un módulo de simulación de una red de datos bajo NS-2 que permita evaluar el desempeño de dos variantes de TCP-por defecto y TCP-Linux bajo condiciones de congestión.

## 2.2 Objetivos Específicos:

1. Modelar una red de datos con tráfico TCP, sometida a condiciones de congestión.

2. Simular en NS-2 el modelo de la red anterior mediante el desarrollo del programa en OTcl que permita incorporar las dos variantes TCP-por defecto y la variante TCP-Linux.

3. Evaluar el desempeño de la red para cada variante mediante el análisis de la ventana de congestión y del Throughput.

4. Elaborar un documento presentando los resultados obtenidos.

5. Elaborar un artículo presentando los resultados más relevantes.

# 3. METODOLOGÍA

Para la realización del trabajo de grado se llevaron a cabo diferentes etapas que permitieron cumplir con los objetivos planteados. A continuación se describen cada una de estas etapas con las actividades realizadas durante el desarrollo de cada una de ellas.

## 3.1 Investigación y Documentación

La etapa de investigación y documentación de los temas relacionados con el software de simulación Network Simulator 2 (NS-2) y el protocolo de transporte TCP (Transmission Control Protocol) fue desarrollada durante toda la ejecución del trabajo de grado con el fin de adquirir conocimientos y herramientas que facilitaran el desarrollo de las actividades planteadas.

Con el fin de sistematizar el uso de la información en la etapa de diseño y desarrollo se organiza la información recopilada de la siguiente manera:

- Generalidades NS-2
- Instalación y ejecución de NS-2
- Lenguaje OTcl
- Generalidades Protocolo TCP
- Algoritmos de Control de Congestión
- Variantes TCP
- Linux TCP

## 3.2 Diseño e Implementación

Esta etapa consiste en el diseño e implementación de la topología que modela la red de datos con tráfico TCP sometida a condiciones de congestión para estudiar los algoritmos de control de congestión en las variantes TCP seleccionadas en NS-2 y en el módulo de NS-2 TCP Linux. A continuación se presenta el desarrollo del programa en OTcl donde se muestra la configuración de cada una de las etapas necesarias para la implementación de la red.

### 3.2.1 Topología

Inicialmente se consideró el desarrollo de una topología de red con varios nodos emisores y receptores, sin embargo con el fin de analizar el comportamiento de las variantes TCP de una forma más detallada, se considera una topología de red más sencilla como la que se muestra en la Figura 3.1

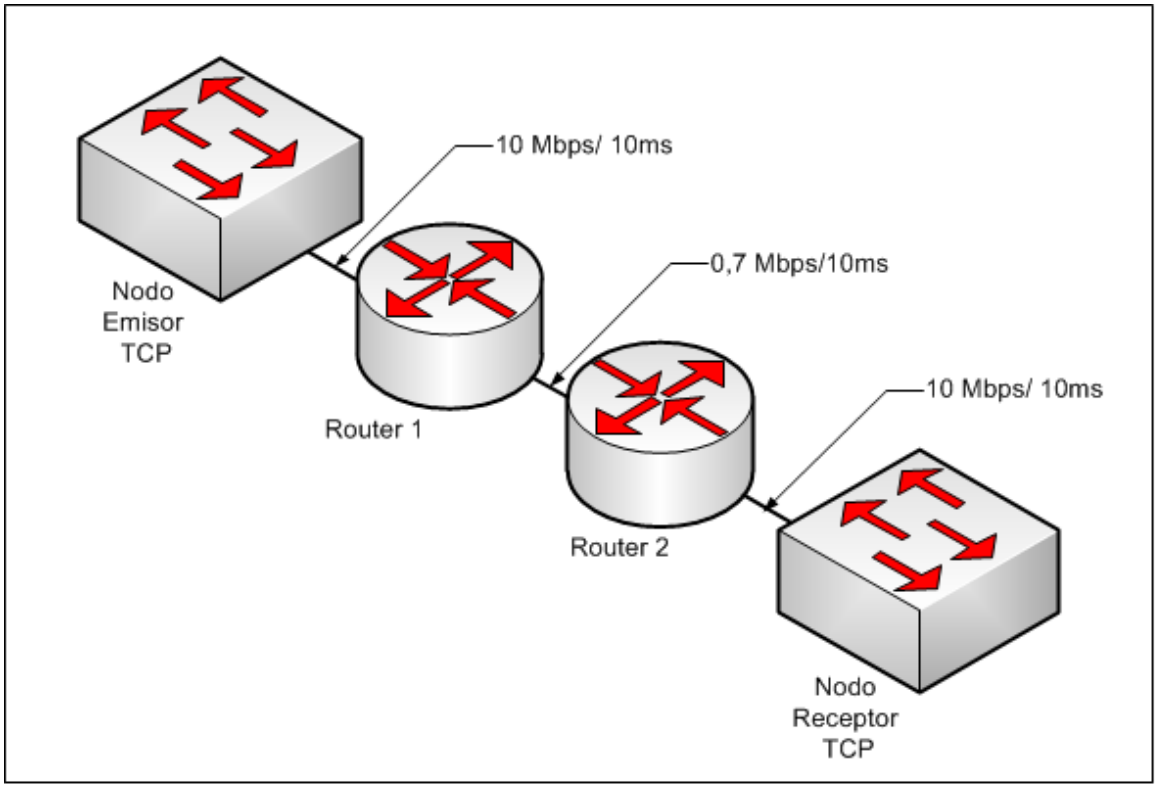


**Figura 3.1 Topología de la Red**

Esta topología es el sistema más sencillo para estudiar la congestión ya que el enlace entre los dos routers (Router1 y Router2) actúa como cuello de botella (*bottleneck*), el cual debe tener un ancho de banda menor que los otros enlaces para que se produzca congestión. En la literatura esta topología se usa comúnmente para estudiar el rendimiento y las variantes del protocolo TCP.

A continuación se presentan los parámetros de diseño de la topología de red implementada para la evaluación de las dos variantes del protocolo TCP:

- Capacidad del Buffer: 20
- Protocolo: TCP
- Aplicación: FTP
- Tamaño del Paquete: 512 bytes
- Tipo de Cola: Droptail

#### *3.2.1.1 Nodos*

La configuración de los nodos de la topología se presenta en la Figura 3.2. Mediante la variable `Nodos` de la línea 1 se determina la cantidad de nodos a configurar; en este caso se configura 1 Nodo Emisor y 1 Nodo Receptor como se muestra en la Figura 3.1.

La creación de los nodos en NS-2 se realiza usando el objeto OTcl `instproc` (procedimiento de instancia): `node`. El nodo emisor se configura de la línea 2 a 5 y el nodo receptor de la línea 6 a 9.

```
#**********************TOPOLOGÍA*****************************
1     set Nodos 1
# Creación de Nodos Fuente: F (Emisores)
2     for {set i 1} {$i <= $Nodos} {incr i} {
3           set F($i) [$ns node]
4           $F($i) color red
5     }
# Creación de Nodos Destino: D (Receptores)
6     for {set i 1} {$i <= $Nodos} {incr i} {
7           set D($i) [$ns node]
8           $D($i) color red
9     }
```

**Figura 3.2 Creación Nodo Emisor y Receptor**

#### *3.2.1.2 Enlaces*

La creación de un enlace unidireccional o bidireccional en NS-2 se realiza usando el `instproc`:

```
simplex-link {src dst bw delay qtype}
duplex-link {src dst bw delay qtype}
```

de la clase `Simulator`, donde `src` es un nodo de inicio, `dst` es un nodo destino, `bw` es el ancho de banda del enlace, `delay` es el retardo de propagación del enlace y `qtype` es el tipo de la cola entre el nodo `src` y el nodo `dst`.

En la Tabla 3.1 se presentan las características de los enlaces de la topología.

**Tabla 3.1 Configuración de los Enlaces**

| Enlace | Ancho de Banda [Mbps] | Retardo [ms] |
|---|---|---|
| 1. Emisor – Router1 | 10 | 10 |
| 2. Router1 – Router2 | 0.7 | 10 |
| 3. Router2 – Receptor | 10 | 10 |

En la Figura 3.3 se presenta la configuración de los nodos enrutadores (línea 10 a la 14) y la creación del enlace cuello de botella que se realiza en la línea 15.

```
# Creación de enrutadores y del enlace cuello de botella.
10    for {set i 1} {$i <= 2} {incr i} {
11          set R($i) [$ns node]
12          $R($i) shape hexagon
13          $R($i) color green
14    }
15    $ns duplex-link $R(1) $R(2) 0.7Mb 10ms  DropTail
```

**Figura 3.3 Configuración Enlace Cuello de Botella**

En la Figura 3.4 se presenta la configuración entre el enlace cuello de botella y los nodos. La configuración del enlace entre el emisor y el cuello de botella se presenta de la línea 16 a 18. La configuración entre el cuello de botella y el destino se presenta de la línea 19 a 21. En la línea 22 se establece el tamaño de la cola del enlace entre los nodos R(1) y R(2) (routers) para que sea de 20 paquetes.

```
# Creación de enlaces entre Fuentes y el cuello de botella
16    for {set i 1} {$i <= $Nodos} {incr i} {
17          $ns duplex-link $F($i) $R(1) 10Mb 10ms DropTail
18    }

# Creación de enlaces entre el cuello de botella y Destinos
19    for {set i 1} {$i <= $Nodos} {incr i} {
20          $ns duplex-link $R(2) $D($i) 10Mb 10ms DropTail
21    }

# Tamaño del buffer de la cola del cuello de botella
22    $ns queue-limit $R(1) $R(2) 20
```

**Figura 3.4 Configuración Enlaces Cuello de Botella y Nodos**

### 3.2.2 Protocolo y Aplicación

La red configurada hasta el momento consiste de 4 nodos. En este escenario el n0 (F1) transfiere datos a n3 (D1) usando el protocolo de transferencia de archivos FTP sobre el protocolo de transporte TCP/IP. En la Figura 3.5 se muestra la topología de la red en el entorno de NAM.

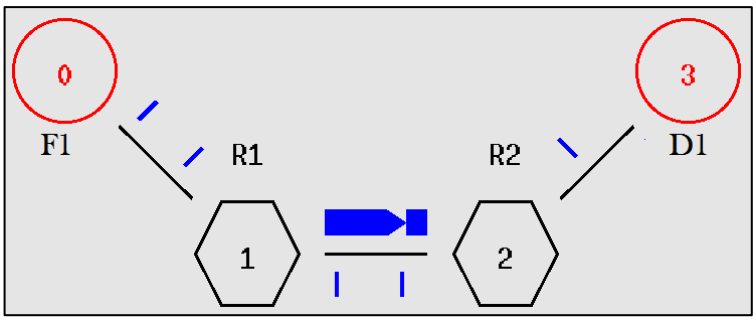


**Figura 3.5 Topología en NAM**

En NS-2 se define el emisor TCP (tcp_F1) mediante la clase OTcl `Agent/TCP` y el receptor TCP (tcp_D1) mediante la clase OTcl `Agent/TCPSink`.

La creación del protocolo (agente) TCP/IP en NS-2 consiste en:

1. Crear un agente de envío, un agente de recepción y una aplicación usando "`new{…}`". Líneas 23 a 30 de la Figura 3.6.

2. Usar el instproc `attach-agent{node agent}` de la clase `Simulator` para crear una conexión entre cada agente y un nodo de entrada. Líneas 32 y 33 de la Figura 3.6.
3. Usar el instproc `connect{src dst}` de la clase `Simulator` para asociar el agente de envío con el agente de recepción. Línea 34 de la Figura 3.6.

En la Figura 3.6 se presenta la configuración del agente TCP descrito en los tres pasos anteriores.

```
#*************** AGENTES DE ENVÍO: EMISORES ****************
23    for {set i 1} {$i <= $Nodos} {incr i} {
24          set tcp_F($i) [new Agent/TCP/Reno]
25          $tcp_F($i) set packetSize_ 512
26    }
27    $tcp_F(1) set fid_ 1

# ********************** AGENTES RECEPTORES*****************
28    for {set i 1} {$i <= $Nodos} {incr i} {
29          set tcp_D($i) [new Agent/TCPSink]
30    }

# ************* CONEXIONES ENTRE EMISOR-RECEPTOR*************
31    for {set i 1} {$i <= $Nodos} {incr i} {
32          $ns attach-agent $F($i) $tcp_F($i)
33          $ns attach-agent $D($i) $tcp_D($i)
34          $ns connect $tcp_F($i) $tcp_D($i)
35    }
```

**Figura 3.6 Creación Agente TCP**

La creación del generador de tráfico FTP en NS-2 se realiza usando la clase `Application/FTP` la cual deriva de la clase `Application`. En la Figura 3.7 se presenta la configuración de la aplicación FTP en la línea 37. En las líneas 38 y 39 se establece el inicio y el fin del generador de tráfico.

```
#*************AGENTES GENERADORES DE TRÁFICO*****************
36    for {set i 1} {$i <= $Nodos} {incr i} {
37          set ftp($i) [$tcp_F($i) attach-source FTP]
38          $ns at 0.0 "$ftp($i) start"
39          $ns at 6.0 "$ftp($i) stop"
40    }
```

**Figura 3.7 Configuración Aplicación FTP**

### 3.2.3 Modelo de Error

La creación del modelo de error en NS-2 se realiza usando la clase OTcl `ErrorModel`. La configuración del modelo de error incluye la configuración de parámetros y de la red. La configuración de parámetros se puede realizar por medio de variables de C++ o de comandos OTcl. La configuración de la red determina donde insertar el modelo de error en un objeto `SimpleLink` [2].

En la Figura 3.8 se presenta la configuración del Modelo de Error implementado en el enlace cuello de botella.

La configuración del modelo de error en NS-2 consiste en:

1. Diseñar y crear una clase de modelo de error en OTcl. Línea 41 de la Figura 3.8.
2. Configurar los parámetros del objeto modelo de error, tales como la probabilidad de error (`rate_`), unidad de error (`unit_`), variable aleatoria (`ranvar_`). Línea 42 y 43 de la Figura 3.8.
3. Insertar un modelo de error en la red usando instproc `lossmodel{lossobj from to}` o instproc `link-lossmodel{lossobj from to}` de la clase `Simulator`. Línea 45 de la Figura 3.8.

```
#******** MODULO DE ERROR EN CUELLO DE BOTELLA**************
41    set moduloE [new ErrorModel]
42    $moduloE set rate_ 0.01
43    $moduloE ranvar [new RandomVariable/Uniform]
44    $moduloE drop-target [new Agent/Null]
45    $ns lossmodel $moduloE $R(1) $R(2)
```

**Figura 3.8 Configuración Modelo de Error**

### 3.2.4 Métricas

Las métricas a observar y analizar en la red diseñada para evaluar el desempeño de la red son la Ventana de Congestión (*cwnd*) y el Rendimiento (*Throughput*).

En la Figura 3.9 se presenta la configuración del procedimiento que permite obtener la ventana de congestión durante la transmisión. De las líneas 47 a 54 se configura el procedimiento llamado plotVentana, que se encarga de registrar cada 0.001s (variable time) el valor de la ventana de congestión en paquetes (`cwnd_`) en el archivo de salida “Ventana” (configuración Línea 46).

```
#Archivos de Salida para guardar cwnd
46    set ventana [open Ventana w]
#Procedimiento para registrar el valor de ventana de congestión
47    proc plotVentana {Fuente_tcp archivo} {
48    global ns
49    set time 0.001
50    set now  [$ns now]
51    set cwnd [$Fuente_tcp set cwnd_]
52    puts $archivo "$now $cwnd"
53    $ns at [expr $now+$time] "plotVentana $Fuente_tcp $archivo"
54    }
#Llamada del Procedimiento.

55    $ns at 0.0 "plotVentana  $tcp_F(1) $ventana"
```

**Figura 3.9 Registro Ventana de Congestión (cwnd)**

En la Figura 3.10 se presenta la configuración del procedimiento que permite determinar el rendimiento. De las líneas 57 a 66 se configura el procedimiento llamado plotThroughput, que se encarga de registrar cada 0.1s el valor de la cantidad de Mbps recibidos por el receptor en el archivo de salida “Throughput”.

```
#Archivos de Salida para guardar el Throughput
56    set rendimiento [open Throughput w]
#Procedimiento para registrar el Throughput
57    proc plot Throughput {Destino_tcp archivo} {
58    global ns
59    set time 0.1
60    set now  [$ns now]
61    set nbytes [$Destino_tcp set bytes_]
62    set throughput [expr ($nbytes * 8.0 / 1000000) / $time]
63    puts  $archivo  "$now $throughput"
64    $Destino_tcp set bytes_ 0
65    $ns at [expr $now+$time] "plotThroughput $Destino_tcp
$archivo"
66    }
#Llamada del Procedimiento.
67    $ns at 0.0 " plotThroughput  $tcp_D(1) $rendimiento"
```

**Figura 3.10 Registro Throughput**

En los procedimientos se obtiene el tiempo de la simulación mediante el insproc `now{}` de la clase `Simulator`. Ambos procedimientos son recursivos, después de cada intervalo de tiempo se llama de nuevo.

### 3.2.5 Archivos

Para el análisis de los resultados se deben filtrar los archivos de salida obtenidos de la simulación que se presentan en la Figura 3.11.

```
set salida      [open out.tr w]
set ventana     [open Ventana w]
set rendimiento [open Throughput w]
```

**Figura 3.11 Archivos de Salida**

El archivo `out.tr` es el generado por el simulador que contiene los eventos que ocurren durante la simulación de la red. El archivo `Ventana` es el generado en el procedimiento para el registro de la ventana de congestión, donde se tiene el valor de cada instante de `cwnd_`. El archivo `Throughput` es el generado en el procedimiento que con base en el registro de los bytes recibidos en el nodo destino calcula el rendimiento de la red.

El filtro del archivo out.tr se realiza mediante el programa AWK de la Figura 3.12 mediante el cual se determinan la cantidad de paquetes transmitidos, perdidos y recibos en la transmisión.

```
BEGIN {
        recibidos = 0;          #paquetes recibidos (r)
        transmitido = 0;        #paquetes transmitidos (-)
        perdido = 0;            #paquetes perdidos (d)
}

{
        evento          = $1;   #Columna del archivo out.tr
        from_Nodo       = $3;
        to_Nodo         = $4;
        size            = $6;
        fuente          = $9;
        destino         = $10;

        if (evento == "r" && from_Nodo==2 && to_Nodo==3 && size == 552 && fuente==0.0 && destino==3.0)
        recibidos++;
        if (from_Nodo==0 && to_Nodo==1 && evento == "-"  && size == 552)
        transmitido++;
        if (evento == "d")
        perdido++;
}
END {
        printf("Paquetes transmitidos: %d\n", transmitido);
        printf("Paquetes recibidos: %d\n", recibidos);
        printf("Paquetes perdidos: %d\n", perdido);
}
```

**Figura 3.12 Script awk para obtener paquetes**

## 3.3 Pruebas

1. Se lleva a cabo la ejecución del script en OTcl y se verifica que el programa esté libre de errores. Inicialmente el tiempo de simulación se determina en 60s, observando que se presenta un patrón igual cada 6s aproximadamente por lo que el tiempo de la duración de simulación para todas las ejecuciones se establece en 6s para facilitar el análisis de los resultados.

2. Se realiza la configuración de la variante TCP en NS-2 y se obtienen los archivos de salida de la ventana de congestión y del rendimiento.

   - Para la variante Reno se ejecuta el script "Reno_NS2.tcl" (Anexo A.1) donde se configura el agente `Agent/TCP/Reno`.
   - Para la variante New Reno se ejecuta el script "NewReno_NS2.tcl" (Anexo A.3) donde se configura el agente `Agent/TCP/Newreno`.

3. Se realiza la configuración de la variante TCP en el módulo de NS-2 Linux TCP y se obtienen los archivos de salida "Ventana" y "Throughput".

   - Para la variante Reno se ejecuta el script "Reno_Linux.tcl" (Anexo A.2) donde se configura el agente `Agent/TCP/Linux`. Mediante el comando `select_ca` se selecciona la variante a implementar en el módulo Linux de NS-2. En la Figura 3.13 se presenta la configuración de la variante Reno mediante `naive_reno`.

```
#TCP Reno Linux
      for {set i 1} {$i <= $Nodos} {incr i} {
            set tcp_F($i) [new Agent/TCP/Linux]
            $tcp_F($i) set timestamps_ true
            $ns at 0 "$tcp_F($i) select_ca naive_reno"
            $tcp_F($i) set packetSize_ 512
      }
```

**Figura 3.13 Selección Variante Reno en Linux**

   - Para la variante New Reno se ejecuta el script "NewReno_Linux.tcl" (Anexo A.4) donde se configura el agente `Agent/TCP/Linux`. En la Figura 3.14 se presenta la configuración de la variante New Reno mediante `reno`.

```
#TCP New Reno Linux
      for {set i 1} {$i <= $Nodos} {incr i} {
            set tcp_F($i) [new Agent/TCP/Linux]
            $tcp_F($i) set timestamps_ true
            $ns at 0 "$tcp_F($i) select_ca reno"
            $tcp_F($i) set packetSize_ 512
      }
```

**Figura 3.14 Selección Variante New Reno en Linux**

4. Se determinan los paquetes transmitidos a partir del archivo out.tr de cada una de las simulaciones mediante el siguiente filtro en awk:

   ```
   >> awk -f '$1=="-" && $3==0 && $4==1 {print $2,23}' out.tr>transmitidos
   ```

   El filtro consiste en comparar las columnas evento, nodo de entrada y nodo de salida del archivo `out.tr` para determinar el tiempo en que se realiza la transmisión de un paquete. Con la sentencia `print` se registra en un archivo llamado `transmitidos` el tiempo de transmisión representado por la columna tiempo del archivo de salida (`$2`). El numero 23 es una variable comodín para la ubicación gráfica de los paquetes transmitidos.
5. Se determinan los paquetes perdidos a partir del archivo out.tr mediante el siguiente filtro en awk:

   ```
   >> awk -f '/d/ {print $2,23}' out.tr > perdidos
   ```

   El filtro consiste en registrar en un archivo llamado `perdidos` los tiempos en los que ocurre una pérdida de paquete.

6. Finalmente para cada variante se llevaron a cabo 3 réplicas diferentes del experimento introduciendo aleatoriedad en la simulación mediante un generador de números aleatorios (*Random number generator (RNG)*).
   Un RNG se inicializa con una semilla (`seed`). La semilla identifica la ubicación inicial en una secuencia-psudo aleatoria, donde un RNG inicia la selección de números [2].

   En la Figura 3.15 se presenta las líneas de código necesarias para introducir aleatoriedad en la simulación.

```
# Replica 1
      $defaultRNG seed 1
      $defaultRNG next-substream

# Replica 2
      $defaultRNG seed 1
      $defaultRNG next-substream
      $defaultRNG next-substream

# Replica 3
      $defaultRNG seed 1
      $defaultRNG next-substream
      $defaultRNG next-substream
      $defaultRNG next-substream
```

**Figura 3.15 Generación de aleatoriedad**

# 4. RESULTADOS Y DISCUSIÓN

En este capítulo se presentan las gráficas de los resultados obtenidos para la ventana de congestión (*cwnd*) y el rendimiento (Throughput) de cada variante de TCP para una de las réplicas realizadas. Los datos fueron tomados de los archivos de traza obtenidos por la simulación de la topología en NS-2; mediante el uso de AWK se realizan filtros sobre los archivos out.tr para determinar los paquetes transmitidos, paquetes enviados, paquetes perdidos, tiempos de retransmisión de los paquetes, entre otros.

## 4.1 Variante TCP Reno

### 4.1.1 TCP Reno por defecto

Descripción Figura 4.1: Línea de color azul representa el valor de la ventana de congestión (*cwnd*) en el tiempo. En la parte superior los puntos de color violeta representan los tiempos de transmisión de los paquetes. La línea discontinua de color negro representa los instantes de pérdida de paquetes. La línea continua de color rojo representa los instantes de retransmisión.

En la Figura 4.1 se muestra el comportamiento de la ventana de congestión (*cwnd*) simulando la versión TCP Reno por defecto de NS-2. Al inicio de la conexión se puede observar un crecimiento rápido de *cwnd* que corresponde a la ejecución del algoritmo de Arranque Lento (*Slow Start*) y mediante la mayor cercanía de los puntos que representan los paquetes transmitidos se puede observar que la velocidad de transmisión del emisor aumenta.

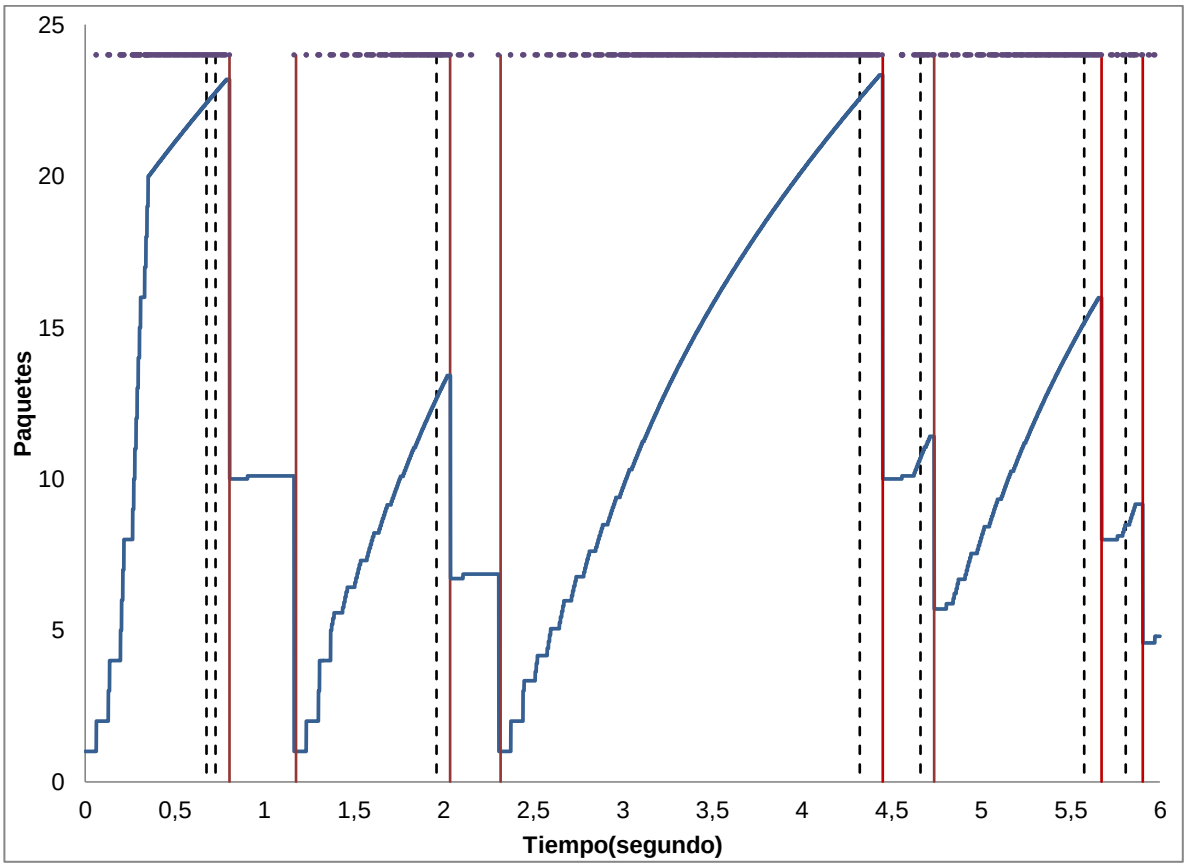


**Figura 4.1 Comportamiento Control de Congestión TCP Reno**

La fase de Slow Start continúa hasta que se presenta congestión en la red determinada por la pérdida de paquetes. En el tiempo 0.6767s se pierde el paquete con número de secuencia 88. En el tiempo 0.8050s se presenta la retransmisión del paquete perdido observando que el valor de *cwnd* se reduce a la mitad del valor de la ventana (window=20) estableciendo el valor del umbral de Slow Start (*ssthresh*=10). La retransmisión del paquete perdido 88 se ejecuta después de que el emisor ha recibido tres ACKs duplicados del paquete con número de secuencia 87, entrando en la fase del algoritmo Retransmisión Rápida / Recuperación Rápida (*Fast Retransmit / Fast Recovery*).

Después de la retransmisión rápida, *cwnd* se sostiene en un valor debido a que durante ese tiempo no hay transmisión de nuevos paquetes por lo que se continua en la fase de *Fast Recovery*.

En el tiempo 1.1764s ocurre un *Timeout*, es decir que el temporizador de retransmisión ha expirado, de esta manera se sale de la fase *Fast Recovery* y *cwnd* se reestablece en 1, iniciando de nuevo el algoritmo de *Slow Start* hasta alcanzar el valor de *ssthresh*. Cuando *cwnd* supera el valor de *ssthresh* se cambia al algoritmo de Prevención de la Congestión (Congestion Avoidance) aumentando *cwnd* de manera lineal, hasta que se detectan pérdidas de datos adicionales.

Después de estos dos eventos de perdida se puede observar que se repite el mismo comportamiento, es decir después de la retransmisión por *Fast Retransmit* en el tiempo 2.0387s el valor de *cwnd* se establece en la mitad y se sale de la fase Fast Recovery cuando ocurre el Timeout en el tiempo 2.3185s momento en el cual se reestablece en 1 la ventana de congestión y se inicia Slow Start.

Las retransmisiones por *Fast Retransmit* en los tiempos 4.7386s, 4.4517s, 5.6731s, 5.9031s establecen el valor de *cwnd* en la mitad de su valor antes de la retransmisión y sale de *Fast Recovery* cuando se recibe un ACK correspondiente a nuevos datos iniciando el algoritmo de *Congestion Avoidance* aumentando linealmente *cwnd.*

En la Tabla 4.1 se presenta el seguimiento de los paquetes perdidos en la conexión con sus tiempos de retransmisión y el valor respectivo de la ventana de congestión, indicando si la retransmisión es realizada por *Fast Retransmit* o por *Timeout*. Estos valores son obtenidos del archivo de traza out.tr de la simulación.

**Tabla 4.1 Retransmisión de Paquetes en TCP Reno**

| Num.Seq Paquete Perdido | Tiempo de Pérdida | Flag | Tiempo de Retransmisión | cwnd | Reenvío después del: | Retransmisión por |
|---|---|---|---|---|---|---|
| 88 | 0,6767 | A | 0,805 | 10 | 3ACK duplicado | *Fast Retransmit* |
| 96 | 0,7272 | A | 1,1764 | 1 | | *Timeout* |
| 188 | 1,9608 | A | 2,0387 | 6,7081 | 3ACK duplicado | *Fast Retransmit* |
| 201 | 2,0365 | A | 2,3185 | 1 | | *Timeout* |
| 477 | 4,3234 | A | 4,4517 | 10 | 3ACK duplicado | *Fast Retransmit* |
| 511 | 4,6624 | A | 4,7386 | 5,7038 | 3ACK duplicado | *Fast Retransmit* |
| 634 | 5,5763 | A | 5,6731 | 7,9933 | 3ACK duplicado | *Fast Retransmit* |
| 660 | 5,808 | A | 5,9031 | 4,5834 | 3ACK duplicado | *Fast Retransmit* |

### 4.1.2 TCP Reno Linux

Descripción Figura 4.2: Línea de color azul representa el valor de la ventana de congestión (*cwnd*) en el tiempo. En la parte superior los puntos de color violeta representan los tiempos de transmisión de los paquetes. La línea discontinua de color negro representa los instantes de pérdida de paquetes. La línea continua de color rojo representa los instantes de retransmisión.

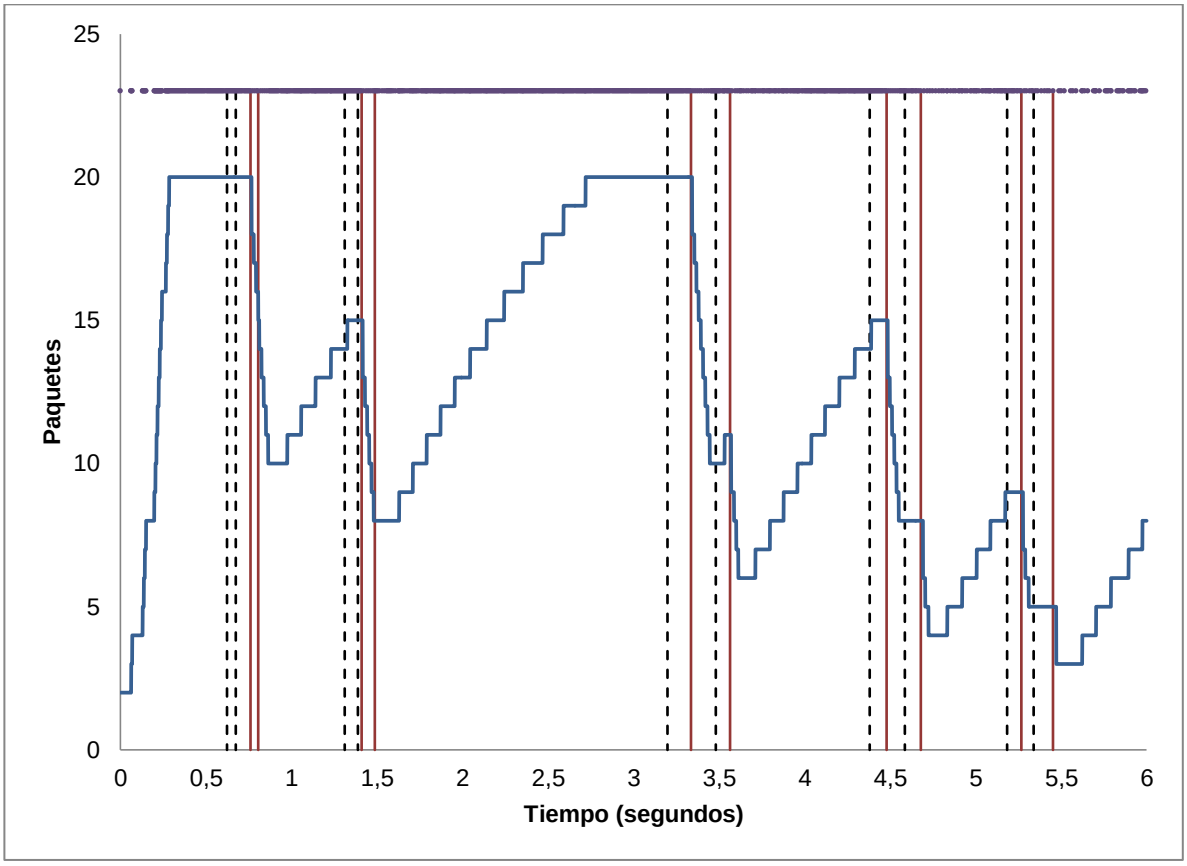


**Figura 4.2 Comportamiento Control de Congestión TCP Reno Linux**

En la Figura 4.2 se muestra el comportamiento de la ventana de congestión (*cwnd*) simulando la versión TCP Linux Reno. La ejecución del algoritmo *Slow Start* se observa solo al inicio de la conexión.

Cuando se detecta la pérdida del paquete 88 por *Fast Retransmit* se inicia la fase del algoritmo *Fast Retransmit / Fast Recovery*, el valor de *cwnd* se reduce gradualmente para evitar más congestión hasta el umbral de arranque lento

(*ssthresh*) que se establece en la mitad del valor de *cwnd* que existía antes de la perdida. Después de que los dos paquetes perdidos son retransmitidos se inicia la fase del algoritmo *Congestion Avoidance* aumentando linealmente la ventana de congestión.

Aquí se evidencia una de las modificaciones y mejoras de Linux TCP en cuanto al algoritmo de *Fast Retransmit/ Fast Recovery* con SACK que ayuda a acelerar la detección de paquetes perdidos, ya que con SACK los ACKs contienen información adicional que hace que el emisor no tenga que esperar un RTT para retransmitir el paquete 96 después de la retransmisión del paquete 88. De esta manera se observa que los Timeouts con la versión Reno por defecto en NS-2 han desaparecido.

En la Tabla 4.2 se presenta el seguimiento de los paquetes perdidos en la conexión con sus tiempos de retransmisión y el valor respectivo de la ventana de congestión, donde se observa que la retransmisión de los paquetes perdidos ya no se realiza después del tercer ACK duplicado sino del cuarto, esto se debe a que algunas implementaciones no estándar como Linux TCP ajusta dinámicamente el umbral de ACK duplicado (*dupthresh*) basado en el reordenamiento observado [6].

**Tabla 4.2 Retransmisión de Paquetes en TCP Linux Reno**

| Num.Seq Paquete Perdido | Tiempo de Pérdida | Flag | Tiempo de Retransmisión | cwnd | Reenvío después de | Retransmisión por |
|---|---|---|---|---|---|---|
| 88 | 0,6219 | A | 0,7592 | 20 | 4ACK duplicado | *Fast Retransmit* |
| 96 | 0,6732 | ---- | 0,8043 | 16 | 12ACK duplicado | *Fast Retransmit* |
| 188 | 1,3101 | A | 1,409 | 15 | 4ACK duplicado | *Fast Retransmit* |
| 201 | 1,3872 | ---- | 1,4861 | 8 | ACK duplicado | *Fast Retransmit* |
| 477 | 3,1974 | A | 3,3348 | 20 | 4ACK duplicado | *Fast Retransmit* |
| 511 | 3,4801 | A | 3,5632 | 11 | 4ACK duplicado | *Fast Retransmit* |
| 634 | 4,3799 | A | 4,4787 | 15 | 4ACK duplicado | *Fast Retransmit* |
| 660 | 4,5854 | A | 4,6787 | 8 | 4ACK duplicado | *Fast Retransmit* |
| 713 | 5,1834 | A | 5,2666 | 9 | 4ACK duplicado | *Fast Retransmit* |
| 728 | 5,3385 | A | 5,4515 | 5 | 4ACK duplicado | *Fast Retransmit* |

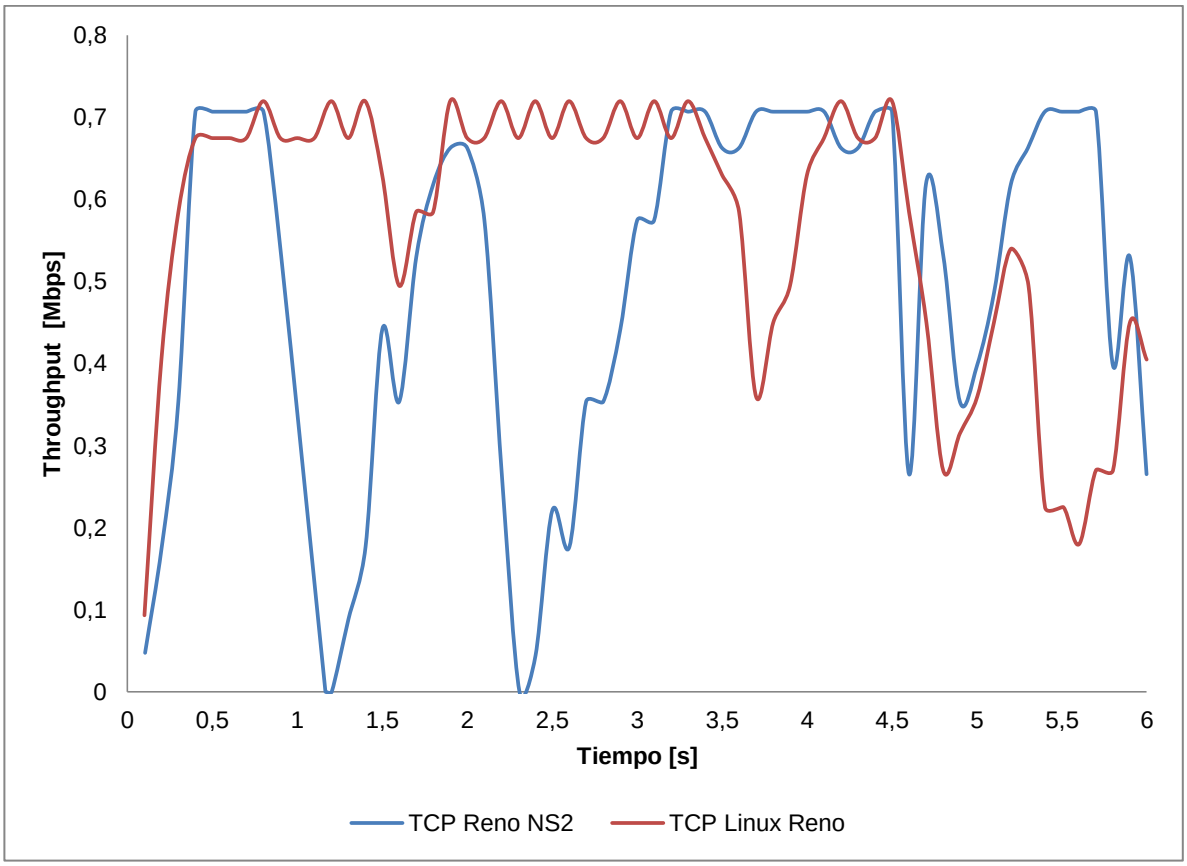


**Figura 4.3 Throughput Variante TCP Reno**

En la Figura 4.3 se presenta el rendimiento (*Throughput*) de ambas implementaciones de la variante Reno. Se puede observar que:

- El *Throughput* aumenta a medida que la carga en la red es mayor en ambas implementaciones, ocurre en la fase de *Slow Start*.
- Cuando hay retransmisión de paquetes el *Throughput* disminuye.
- En la implementación de TCP Reno NS-2 el *throughput* llega a un valor de cero porque existe un *Timeout*. En la Figura 4.1 se puede observar que en ese momento no hay transmisión de paquetes mediante los puntos de la parte superior que indican los tiempos de los paquetes transmitidos.
- El rendimiento de la red en la implementación de la variante TCP Linux Reno es mayor como se puede observar en la Tabla 4.3

**Tabla 4.3 Throughput Variante Reno**

| Variante | Throughput [Mbps] | |
|---|---|---|
| | Promedio | Desviación |
| Reno NS-2 | 0,4902 | 0,2355 |
| Reno Linux | 0,5688 | 0,1668 |

## 4.2 Variante TCP New Reno

### 4.2.1 TCP New Reno por defecto

Descripción Figura 4.4: Línea de color azul representa el valor de la ventana de congestión (*cwnd*) en el tiempo. En la parte superior los puntos de color violeta representan los tiempos de transmisión de los paquetes. La línea discontinua de color negro representa los instantes de pérdida de paquetes. La línea continua de color rojo representa los instantes de retransmisión.

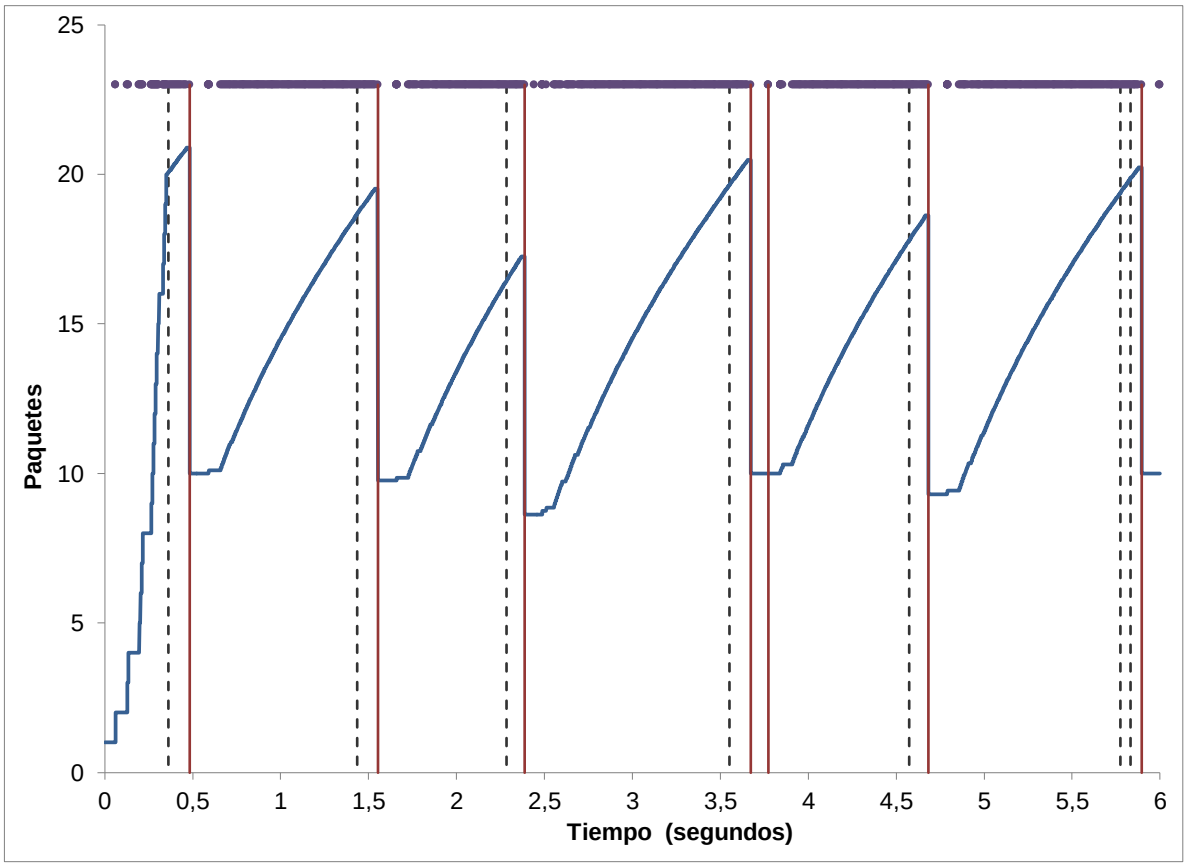


**Figura 4.4 Comportamiento Control de Congestión TCP New Reno**

En la Figura 4.4 se muestra el comportamiento de la ventana de congestión (*cwnd*) simulando la versión TCP New Reno. La ejecución del algoritmo *Slow Start* se observa solo al inicio de la conexión.

La fase de *Slow Start* continúa hasta que se presenta congestión en la red determinada por la pérdida de paquetes. En el tiempo 0.4833s se presenta la retransmisión del paquete perdido observando que el valor de *cwnd* se reduce a la mitad y se establece el valor de *ssthresh*. La retransmisión del paquete perdido 37 se ejecuta después de que el emisor ha recibido tres ACKs duplicados del paquete con número de secuencia 36, entrando en la fase de *Fast Retransmit / Fast Recovery.*

Se observa que durante el estado *Fast Recovery* la ventana de congestión se mantiene en el mismo valor (*cwnd*=10) hasta que en el tiempo 0.5905s se recibe en el emisor el ACK del paquete 56 que representa el Punto de Recuperación de la fase *Fast Retransmit / Fast Recovery*; por tanto termina la fase de *Fast Recovery* y empieza la fase del algoritmo *Congestion Avoidance* evidenciando el aumento lineal de *cwnd* hasta una nueva retransmisión.

Las retransmisiones en los tiempos 0.4833s, 1.5524s, 4.6836 y 5.896s producen el mismo comportamiento de la ventana de congestión descrito anteriormente: se entra en la fase de *Fast Retransmit* al retrasmitir el paquete y al recibir un ACK mayor al Punto de Recuperación de cada retransmisión rápida se termina la fase de *Fast Recovery* y se inicia la fase de *Congestion Avoidance.*

En los tiempos 3.6729s y 3.7739s se retransmiten los dos paquetes perdidos en la ventana sin observar *timeout* como en el caso de la versión Reno cuando existe más de un paquete perdido; esto se debe al registro del punto de recuperación que implementa New Reno que permite retransmitir múltiples paquetes perdidos mediante la llegada al emisor de ACKs parciales (reconocimientos después de la retransmisión).

Cuando el emisor recibe un ACK nuevo, es decir un ACK para un paquete mayor o igual al paquete del punto de recuperación se sale de *Fast Recovery* y empieza *cwnd* aumentar linealmente mediante *Congestion Avoidance* hasta una nueva retransmisión.

En la Tabla 4.4 se registra los tiempos de retransmisión de cada paquete perdido con el valor de *cwnd*. Se registra el paquete con el número de secuencia más alto enviado después del paquete perdido, denominado Punto de Recuperación.

**Tabla 4.4 Retransmisión de Paquetes en TCP New Reno**

| Num.Seq Paquete Perdido | Tiempo de Pérdida | Flag | Tiempo de Retransmisión | cwnd | Reenvío después del: | Retransmisión por | Punto de Recuperación |
|---|---|---|---|---|---|---|---|
| 37 | 0,3613 | A | 0,4833 | 10 | 3ACK duplicado | *Fast Retransmit* | Paquete 56 |
| 196 | 1,4367 | A | 1,5524 | 9,7554 | 3ACK duplicado | *Fast Retransmit* | Paquete 215 |
| 316 | 2,2833 | A | 2,3864 | 8,6276 | 3ACK duplicado | *Fast Retransmit* | Paquete 334 |
| 506 | 3,5509 | A | 3,6729 | 10 | 3ACK duplicado | *Fast Retransmit* | Paquete 525 |
| 512 | 3,5887 | ---- | 3,7739 | 10 | ACK del paquete 511 | *ACK Parcial* | |
| 648 | 4,5741 | A | 4,6836 | 9,3089 | 3ACK duplicado | *Fast Retransmit* | Paquete 667 |
| 828 | 5,7740 | A | 5,8961 | 10 | 3ACK duplicado | *Fast Retransmit* | Paquete 847 |
| 837 | 5,8308 | ---- | El paquete no se recupera porque se encuentra al final de la simulación. | | | | |

### 4.2.2 TCP New Reno Linux

Descripción Figura 4.5: Línea de color azul representa el valor de la ventana de congestión (*cwnd*) en el tiempo. En la parte superior los puntos de color violeta representan los tiempos de transmisión de los paquetes. La línea discontinua de color negro representa los instantes de pérdida de paquetes. La línea continua de color rojo representa los instantes de retransmisión.

Cuando el emisor recibe ACKs de paquetes nuevos, es decir que no recibe un ACK parcial o menor al punto de recuperación se sale de la fase *Fast Recovery* y se observa el aumento lineal de *cwnd* gracias al algoritmo *Congestion Avoidance*.

Igual que en la implementación anterior (New Reno por defecto en NS-2) se observa que la retransmisión de varios paquetes perdidos se puede realizar sin observar un restablecimiento de *cwnd*. Sin embargo, se observa que la segunda retransmisión (t=3.5138s) se realiza después de 0.0323s de la primera retransmisión, es decir en un tiempo más corto que en la implementación de New Reno anterior (0.1009s)

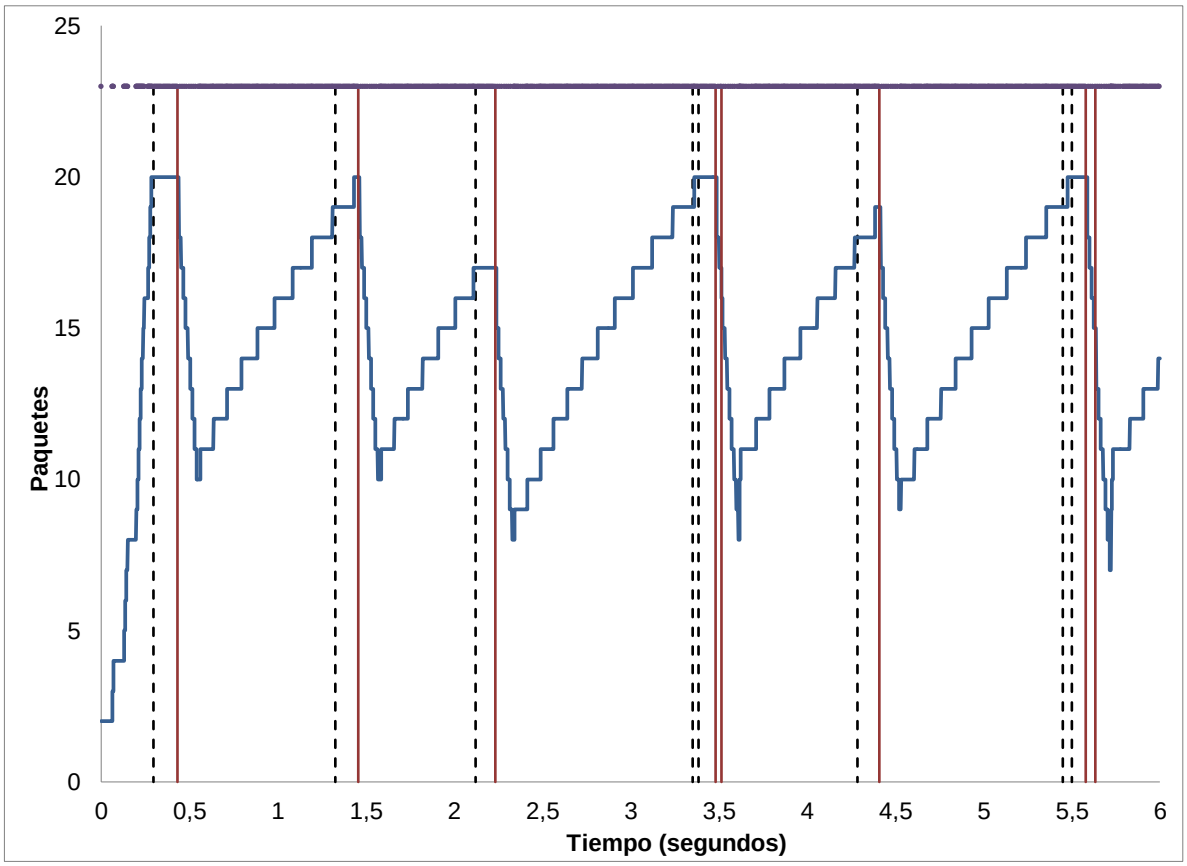


**Figura 4.5 Comportamiento Control de Congestión TCP New Reno Linux**

En la Tabla 4.5 se presenta el seguimiento de los paquetes perdidos en la conexión con sus tiempos de retransmisión y el valor respectivo de la ventana de congestión, donde se observa que la retransmisión de los paquetes perdidos ya no necesariamente se realiza después del tercer ACK duplicado sino del cuarto.

**Tabla 4.5 Retransmisión de Paquetes en TCP Linux New Reno**

| Num.Seq Paquete Perdido | Tiempo de Pérdida | Flag | Tiempo de Retransmisión | cwnd | Reenvío | Retransmisión por | Punto de Recuperación |
|---|---|---|---|---|---|---|---|
| 37 | 0,2947 | A | 0,4317 | 20 | 4ACK duplicado | *Fast Retransmit* | Paquete 59 |
| 196 | 1,3257 | A | 1,4566 | 20 | 4ACK duplicado | *Fast Retransmit* | Paquete 218 |
| 316 | 2,1194 | A | 2,2311 | 17 | 4ACK duplicado | *Fast Retransmit* | Paquete 335 |
| 506 | 3,3506 | A | 3,4815 | 20 | 3ACK duplicado | *Fast Retransmit* | Paquete 528 |
| 512 | 3,3827 | | 3,5138 | 17 | 9ACK duplicado | *ACK Parcial* | |
| 648 | 4,2855 | A | 4,4100 | 19 | 4ACK duplicado | *Fast Retransmit* | Paquete 669 |
| 828 | 5,4489 | A | 5,5798 | 20 | 4ACK duplicado | *Fast Retransmit* | Paquete 850 |
| 837 | 5,5002 | | 5,6313 | 15 | 12ACK duplicado | *ACK Parcial* | |

En la Figura 4.6 se presenta el rendimiento (*Throughput*) de ambas implementaciones de la variante New Reno. Se puede observar que:

- El *Throughput* aumenta a medida que la carga en la red es mayor en ambas implementaciones.
- Cuando hay retransmisión de paquetes el *Throughput* disminuye. Se observa que el rendimiento en TCP Linux New Reno presenta una disminución menor en cada retransmisión que la variante New Reno en NS-2 por defecto.
- El *throughput* no llega a un valor de cero porque los eventos de *Timeout* han desaparecido por la mejora del algoritmo de *Fast Recovery*.
- El rendimiento de la red en la implementación de la variante TCP Linux New Reno es mayor como se puede observar en la Tabla 4.6

**Tabla 4.6 Throughput Variante New Reno**

| Variante | Throughput [Mbps] | |
|---|---|---|
| | Promedio | Desviación |
| New Reno NS-2 | 0,6183 | 0,1513 |
| New Reno Linux | 0,6655 | 0,0906 |

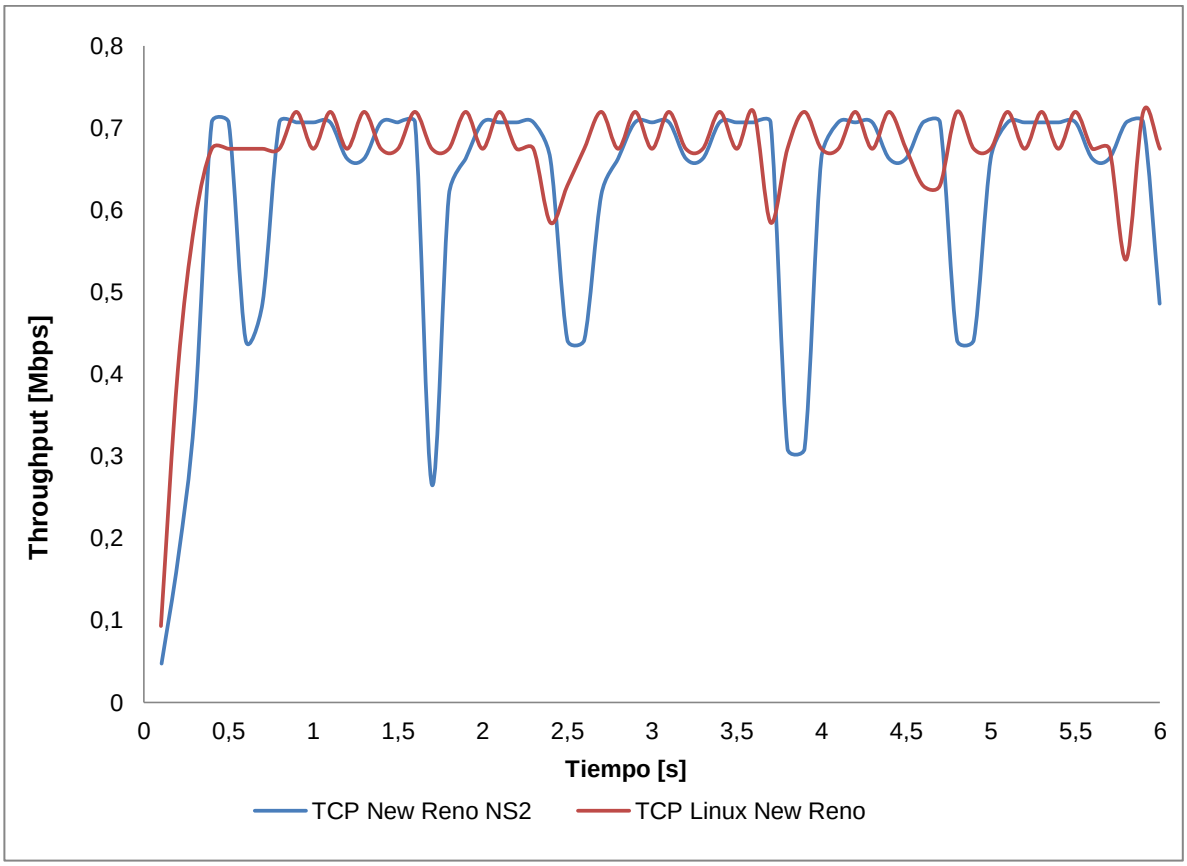


**Figura 4.6 Throughput Variante TCP New Reno**

La variante New Reno en ambas implementaciones presenta mejores resultados que la variante Reno, debido a la capacidad que tiene de retransmitir más de un paquete perdido en una ventana gracias al Punto de Recuperación que adiciona en su algoritmo de *Fast Recovery*.

La implementación de la variante New Reno en el módulo NS-2 TCP Linux presenta un mejor rendimiento debido a que la desviación del Throughput es menor.

Después de obtener los resultados de la simulación, se elabora un artículo presentando los resultados (Anexo E).

# 5. CONCLUSIONES

Network Simulator 2 como herramienta de simulación permite al usuario implementar escenarios para modelar redes de comunicación con diferentes protocolos, aplicaciones, enlaces y elementos de red. Gracias a su distribución libre ha permitido el desarrollo de nuevos protocolos y algoritmos que representan mejoras en el rendimiento de la red. La gran cantidad de scripts de código abierto que se encuentran en la literatura y en la web pueden ser libremente accedidos y modificados, permitiendo al usuario tener un proceso de aprendizaje y manejo de la herramienta con mayor facilidad.

El desarrollo del módulo de simulación de la red de datos en NS-2 se realiza mediante un script en lenguaje OTcl incorporando los elementos de red, enlaces, protocolos y aplicaciones de una topología sencilla sometida a congestión que permite evaluar el desempeño de los algoritmos de control de congestión de las variantes Reno y New Reno del protocolo de TCP.

El modelo de la red de datos con tráfico TCP se puede someter a condiciones de congestión mediante el cambio de la capacidad del buffer o mediante la inclusión de un Modelo de Error con una probabilidad de error en la trasmisión de paquetes.

Las características adicionales de los algoritmos de control de congestión implementados en Linux-TCP permiten tener mejoras en el rendimiento de la red y la detección de la congestión se realiza de una forma más rápida y precisa. Mediante la ventana de congestión (*cwnd*) se puede observar que en las implementaciones de la variante TCP en el módulo NS-2 TCP Linux, las retransmisiones de los paquetes perdidos se realizan más rápido y por tanto el rendimiento (*throughput*) de la red es mayor.

Los algoritmos de control de congestión Arranque Lento (*Slow Start*), Prevención de Congestión (*Congestion Avoidance*), Retransmisión Rápida (*Fast Retransmit*) y Recuperación Rápida (*Fast Recovery*) presentan el comportamiento básico del control de congestión AIMD (*Additive Increase / Multiplicative Decrease)* tanto para la implementación de la variante TCP en NS-2 y en la implementación en el módulo NS-2 TCP Linux.

# 6. RECOMENDACIONES Y TRABAJOS FUTUROS

- Desarrollar una topología de red posiblemente con mayor cantidad de nodos que permita implementar otros protocolos y aplicaciones soportados en NS-2.

- Implementar diferentes variantes del protocolo TCP que se pueden simular en NS-2 y que el módulo NS-2 TCP Linux soporta para estudiar los algoritmos de control de congestión implementados en ellas.

- Cambiar los diferentes parámetros de configuración de la red como la velocidad y retardo de los enlaces de la topología, para evaluar el desempeño de la red.

- Implementar la topología de red en la nueva versión de Network Simulator NS3 que utiliza el lenguaje de programación C++ para comparar los resultados obtenidos del comportamiento de los algoritmos de control de congestión de cada variante en NS-2.

# 7. REFERENCIAS


[1] E. ALTMAN y T. JIMENEZ, NS Simulator for Beginners, Morgan & Claypool, 2012, p. 184.

[2] T. ISSARIYAKUL y E. HOSSAIN, Introduction to Network Simulator NS2, Springer, 2009, p. 438.

[3] K. FALL y K. VARADHAN, The ns Manual (formerly ns Notes and Documentation), 2011, p. 434.

[4] J. F. KUROSE y K. W. ROSS, Redes de Computadores. Un enfoque descendente basado en Internet., 2 ed.

[5] L. L. PETERSON y B. S. DAVIE, Computer Networks, a systems approach, 5 ed., Elsevier, 2012, p. 921.

[6] W. R. STEVENS, TCP/IP Illustrated, Volume 1: The Protocols, Segunda ed., Addison Wesley.

[7] C. Callegari, S. Giordano, M. Pagano y T. Pepe, «Behavior analysis of TCP Linux variants,» *Computer Networks,* pp. 462-476, October 2011.

[8] S. KITTIPERACHOL, Z. SUN y H. CRUICKSHANK, «Integration of Linux TCP Implementation and Simulation: Verification and Validation,» pp. 413-420, June 2008.

[9] D. X. WEI y P. CAO, «NS-2 TCP-Linux: an NS-2 TCP implementation with congestion control algorithms from Linux,» *Proceeding of 2006 workshop of ns-2: the IP network simulator,* p. 10, October 2006.

[10] X. D. WEI, «A mini-tutorial for TCP-Linux in NS-2,» [En línea]. Available: http://netlab.caltech.edu/projects/ns2tcplinux/ns2linux/index.html.

[11] P. SAROLAHTI y A. KUZNETSOV, «Congestion Control in Linux TCP,» *USENIX Annual Technical Conference,* pp. 49-62, 2002.

# ANEXOS

## ANEXO A. Script Variante Reno por defecto en NS-2 (Reno_NS2.tcl)

```
#             N0                N3
#   10Mb/10ms \   0.7Mb/10ms   / 10Mb/10ms
#              N1 ----------- N2

# Creación objeto simulador
      set ns [new Simulator]
      $ns color 1 blue
      $ns color 2 red
# Replica 1
      $defaultRNG seed 1
      $defaultRNG next-substream

# ************** Creación archivo de traza**************
      set traza1 [open red.nam w]
      $ns namtrace-all $traza1
      set salida [open out.tr w]
#Archivos de Salida
      set ventana [open Ventana w]
      set rendimiento [open Throughput w]
      $ns trace-all $salida

# Procedimiento fin: Se cierra archivo de traza y se ejecuta NAM
      proc fin { } {
            global ns traza1 salida
            $ns flush-trace
            close $traza1
            close $salida
            exec nam red.nam &
            exit 0
       }
#**********************TOPOLOGÍA*****************************
set Nodos 1
# Creación de Nodos Fuente: F (Emisores)
      for {set i 1} {$i <= $Nodos} {incr i} {
            set F($i) [$ns node]
            $F($i) color red
      }
# Creación de enrutadores y del enlace cuello de botella.
      for {set i 1} {$i <= 2} {incr i} {
            set R($i) [$ns node]
            $R($i) shape hexagon
            $R($i) color green
      }
      $ns duplex-link $R(1) $R(2) 0.7Mb 10ms  DropTail
```

```
# Creación de Nodos Destino: D (Receptores)
      for {set i 1} {$i <= $Nodos} {incr i} {
            set D($i) [$ns node]
            $D($i) color red
      }
# Creación de enlaces entre Fuentes y el cuello de botella
      for {set i 1} {$i <= $Nodos} {incr i} {
            $ns duplex-link $F($i) $R(1) 10Mb 10ms DropTail
      }
# Creación de enlaces entre el cuello de botella y Destinos
      for {set i 1} {$i <= $Nodos} {incr i} {
            $ns duplex-link $R(2) $D($i) 10Mb 10ms DropTail
      }
# Tamaño del buffer de la cola del cuello de botella
      $ns queue-limit $R(1) $R(2) 20

# Orientación de los nodos para NAM
      $ns duplex-link-op $F(1) $R(1) orient right-down
      $ns duplex-link-op $R(1) $R(2) orient right
      $ns duplex-link-op $R(2) $D(1) orient right-up
# Etiquetas
      $ns at 0.0 "$R(1) label \"R1\" "
      $ns at 0.0 "$R(2) label \"R2\" "
      $ns at 0.0 "$F(1) label \"F1\" "
      $ns at 0.0 "$D(1) label \"D1\" "

#******** MODULO DE ERROR EN CUELLO DE BOTELLA**************
      set moduloE [new ErrorModel]
      $moduloE set rate_ 0.01
      $moduloE ranvar [new RandomVariable/Uniform]
      $moduloE drop-target [new Agent/Null]
      $ns lossmodel $moduloE $R(1) $R(2)

#*************** AGENTES DE ENVÍO: EMISORES ***************
      for {set i 1} {$i <= $Nodos} {incr i} {
            set tcp_F($i) [new Agent/TCP/Reno]
            $tcp_F($i) set packetSize_ 512
      }
      $tcp_F(1) set fid_ 1
# ********************** AGENTES RECEPTORES*****************
      for {set i 1} {$i <= $Nodos} {incr i} {
            set tcp_D($i) [new Agent/TCPSink]
      }
# ************* CONEXIONES ENTRE EMISOR-RECEPTOR*************
      for {set i 1} {$i <= $Nodos} {incr i} {
      # Agentes fuentes TCP asociados a los nodos emisores
            $ns attach-agent $F($i) $tcp_F($i)
      # Destinos TCP se anexan a los nodos receptores
            $ns attach-agent $D($i) $tcp_D($i)
      # Conexión de cada pareja
            $ns connect $tcp_F($i) $tcp_D($i)
}
```

```
#*************AGENTES GENERADORES DE TRÁFICO*****************
      for {set i 1} {$i <= $Nodos} {incr i} {
            set ftp($i) [$tcp_F($i) attach-source FTP]
            $ns at 0.0 "$ftp($i) start"
            $ns at 6.0 "$ftp($i) stop"
      }
#*******************REGISTRO*********************************

proc plotVentana {Fuente_tcp archivo} {
global ns
set time 0.001
#Obtener el valor del tiempo actual
set now  [$ns now]
#Obtener el valor de la ventana de congestión
set cwnd [$Fuente_tcp set cwnd_]
#Registrar en el fichero el par de valores
puts $archivo "$now $cwnd"
$ns at [expr $now+$time] "plotVentana $Fuente_tcp $archivo"
}

proc plotThroughput {Destino_tcp archivo} {
global ns
set time 0.1
set now    [$ns now]
set nbytes [$Destino_tcp set bytes_]
set throughput [expr ($nbytes * 8.0 / 1000000) / $time]
puts  $archivo  "$now $throughput"
$Destino_tcp set bytes_ 0
$ns at [expr $now+$time] "plotThroughput $Destino_tcp $archivo"
}
#*********************************************************
# Eventos
      $ns at 0.0 "plotVentana      $tcp_F(1) $ventana"
      $ns at 0.0 "plotThroughput   $tcp_D(1) $rendimiento"
      $ns at 6.0 "fin"

# Comenzar la simulación
      $ns run
```

## ANEXO B. Script Variante Reno en NS-2 TCP Linux (Reno_Linux.tcl)

```
# Creación objeto simulador
      set ns [new Simulator]
      $ns color 1 blue
      $ns color 2 red
# Replica 2
      $defaultRNG seed 1
      $defaultRNG next-substream
      $defaultRNG next-substream
# ************** Creación archivo de traza**************
      set traza1 [open red.nam w]
      $ns namtrace-all $traza1
      set salida [open out.tr w]

#Archivos de Salida
      set ventana [open Ventana w]
      set rendimiento [open Throughput w]
      $ns trace-all $salida
# Procedimiento fin: cierra archivo de traza y se ejecuta NAM
      proc fin { } {
            global ns traza1 salida
            $ns flush-trace
            close $traza1
            close $salida
            exec nam red.nam &
            exit 0
       }
#**********************TOPOLOGÍA*****************************
set Nodos 1
# Creación de Nodos Fuente: F (Emisores)
      for {set i 1} {$i <= $Nodos} {incr i} {
            set F($i) [$ns node]
            $F($i) color red
      }
# Creación de enrutadores y del enlace cuello de botella.
      for {set i 1} {$i <= 2} {incr i} {
            set R($i) [$ns node]
            $R($i) shape hexagon
            $R($i) color green
      }
      $ns duplex-link $R(1) $R(2) 0.7Mb 10ms  DropTail

# Creación de Nodos Destino: D (Receptores)
      for {set i 1} {$i <= $Nodos} {incr i} {
            set D($i) [$ns node]
            $D($i) color red
      }
# Creación de enlaces entre Fuentes y el cuello de botella
      for {set i 1} {$i <= $Nodos} {incr i} {
            $ns duplex-link $F($i) $R(1) 10Mb 10ms DropTail
      }
```

```
# Creación de enlaces entre el cuello de botella y Destinos
      for {set i 1} {$i <= $Nodos} {incr i} {
            $ns duplex-link $R(2) $D($i) 10Mb 10ms DropTail
      }
# Tamaño del buffer de la cola del cuello de botella
      $ns queue-limit $R(1) $R(2) 20

# Orientación de los nodos para NAM
      $ns duplex-link-op $F(1) $R(1) orient right-down
      $ns duplex-link-op $R(1) $R(2) orient right
      $ns duplex-link-op $R(2) $D(1) orient right-up

# Etiquetas
      $ns at 0.0 "$R(1) label \"R1\" "
      $ns at 0.0 "$R(2) label \"R2\" "
      $ns at 0.0 "$F(1) label \"F1\" "
      $ns at 0.0 "$D(1) label \"D1\" "

#******** MODULO DE ERROR EN CUELLO DE BOTELLA**************
      set moduloE [new ErrorModel]
      $moduloE set rate_ 0.01
      $moduloE ranvar [new RandomVariable/Uniform]
      $moduloE drop-target [new Agent/Null]
      $ns lossmodel $moduloE $R(1) $R(2)

#*************** AGENTES DE ENVÍO: EMISORES ***************
      for {set i 1} {$i <= $Nodos} {incr i} {
            set tcp_F($i) [new Agent/TCP/Linux]
            $tcp_F($i) set timestamps_ true
            $ns at 0 "$tcp_F($i) select_ca naive_reno"
            $tcp_F($i) set packetSize_ 512
      }
      $tcp_F(1) set fid_ 1

# ********************** AGENTES RECEPTORES*****************
      for {set i 1} {$i <= $Nodos} {incr i} {
            set tcp_D($i) [new Agent/TCPSink]
      }
# ************* CONEXIONES ENTRE EMISOR-RECEPTOR*************
      for {set i 1} {$i <= $Nodos} {incr i} {
            $ns attach-agent $F($i) $tcp_F($i)
            $ns attach-agent $D($i) $tcp_D($i)
            $ns connect $tcp_F($i) $tcp_D($i)
}
#*************AGENTES GENERADORES DE TRÁFICO*****************
      for {set i 1} {$i <= $Nodos} {incr i} {
            set ftp($i) [$tcp_F($i) attach-source FTP]
            $ns at 0.0 "$ftp($i) start"
            $ns at 6.0 "$ftp($i) stop"
      }
```

```
#*******************REGISTRO*********************************
proc plotVentana {Fuente_tcp archivo} {
global ns
set time 0.001
set now  [$ns now]
set cwnd [$Fuente_tcp set cwnd_]
puts $archivo "$now $cwnd"
$ns at [expr $now+$time] "plotVentana $Fuente_tcp $archivo"
}

proc plotThroughput {Destino_tcp archivo} {
global ns
set time 0.1
set now    [$ns now]
set nbytes [$Destino_tcp set bytes_]
set throughput [expr ($nbytes * 8.0 / 1000000) / $time]
puts  $archivo  "$now $throughput"
$Destino_tcp set bytes_ 0
$ns at [expr $now+$time] "plotThroughput $Destino_tcp $archivo"
}
#*****************Eventos**********************************
      $ns at 0.0 "plotVentana      $tcp_F(1) $ventana"
      $ns at 0.0 "plotThroughput   $tcp_D(1) $rendimiento"
      $ns at 6.0 "fin"

# Comenzar la simulación
      $ns run
```

## ANEXO C. Script Variante New Reno por defecto en NS-2 (NewReno_NS2.tcl)

```
# Creación objeto simulador
      set ns [new Simulator]
      $ns color 1 blue
      $ns color 2 red
# Replica 3
      $defaultRNG seed 1
      $defaultRNG next-substream
      $defaultRNG next-substream
      $defaultRNG next-substream

# ************** Creación archivo de traza**************
      set traza1 [open red.nam w]
      $ns namtrace-all $traza1
      set salida [open out.tr w]
#Archivos de Salida
      set ventana [open Ventana w]
      set rendimiento [open Throughput w]
      $ns trace-all $salida
# Procedimiento fin: cierra archivo de traza y se ejecuta NAM
      proc fin { } {
            global ns traza1 salida
            $ns flush-trace
            close $traza1
            close $salida
            exec nam red.nam &
            exit 0
       }
#**********************TOPOLOGÍA*****************************
set Nodos 1
# Creación de Nodos Fuente: F (Emisores)
      for {set i 1} {$i <= $Nodos} {incr i} {
            set F($i) [$ns node]
            $F($i) color red
      }
# Creación de enrutadores y del enlace cuello de botella.
      for {set i 1} {$i <= 2} {incr i} {
            set R($i) [$ns node]
            $R($i) shape hexagon
            $R($i) color green
      }
      $ns duplex-link $R(1) $R(2) 0.7Mb 10ms  DropTail
# Creación de Nodos Destino: D (Receptores)
      for {set i 1} {$i <= $Nodos} {incr i} {
            set D($i) [$ns node]
            $D($i) color red
      }
# Creación de enlaces entre Fuentes y el cuello de botella
      for {set i 1} {$i <= $Nodos} {incr i} {
            $ns duplex-link $F($i) $R(1) 10Mb 10ms DropTail
      }
```

```
# Creación de enlaces entre el cuello de botella y Destinos
      for {set i 1} {$i <= $Nodos} {incr i} {
            $ns duplex-link $R(2) $D($i) 10Mb 10ms DropTail
      }
# Tamaño del buffer de la cola del cuello de botella
      $ns queue-limit $R(1) $R(2) 20

# Orientación de los nodos para NAM
      $ns duplex-link-op $F(1) $R(1) orient right-down
      $ns duplex-link-op $R(1) $R(2) orient right
      $ns duplex-link-op $R(2) $D(1) orient right-up
# Etiquetas
      $ns at 0.0 "$R(1) label \"R1\" "
      $ns at 0.0 "$R(2) label \"R2\" "
      $ns at 0.0 "$F(1) label \"F1\" "
      $ns at 0.0 "$D(1) label \"D1\" "

#******** MODULO DE ERROR EN CUELLO DE BOTELLA**************
      set moduloE [new ErrorModel]
      $moduloE set rate_ 0.01
      $moduloE ranvar [new RandomVariable/Uniform]
      $moduloE drop-target [new Agent/Null]
      $ns lossmodel $moduloE $R(1) $R(2)

#*************** AGENTES DE ENVÍO: EMISORES ***************
      for {set i 1} {$i <= $Nodos} {incr i} {
            set tcp_F($i) [new Agent/TCP/Newreno]
            $tcp_F($i) set packetSize_ 512
      }
      $tcp_F(1) set fid_ 1


# ********************** AGENTES RECEPTORES*****************
      for {set i 1} {$i <= $Nodos} {incr i} {
            set tcp_D($i) [new Agent/TCPSink]
      }
# ************* CONEXIONES ENTRE EMISOR-RECEPTOR*************
      for {set i 1} {$i <= $Nodos} {incr i} {
      # Agentes fuentes TCP asociados a los nodos emisores
            $ns attach-agent $F($i) $tcp_F($i)
      # Destinos TCP se anexan a los nodos receptores
            $ns attach-agent $D($i) $tcp_D($i)
      # Conexión de cada pareja
            $ns connect $tcp_F($i) $tcp_D($i)
}
#*************AGENTES GENERADORES DE TRÁFICO*****************
      for {set i 1} {$i <= $Nodos} {incr i} {
            set ftp($i) [$tcp_F($i) attach-source FTP]
            $ns at 0.0 "$ftp($i) start"
            $ns at 6.0 "$ftp($i) stop"
      }
```

```
#******************REGISTRO*********************************
proc plotVentana {Fuente_tcp archivo} {
global ns
set time 0.001
#Obtener el valor del tiempo actual
set now  [$ns now]
#Obtener el valor de la ventana de congestión
set cwnd [$Fuente_tcp set cwnd_]
#Registrar en el fichero el par de valores
puts $archivo "$now $cwnd"
$ns at [expr $now+$time] "plotVentana $Fuente_tcp $archivo"
}

proc plotThroughput {Destino_tcp archivo} {
global ns
set time 0.1
set now    [$ns now]
set nbytes [$Destino_tcp set bytes_]
set throughput [expr ($nbytes * 8.0 / 1000000) / $time]
puts  $archivo  "$now $throughput"
$Destino_tcp set bytes_ 0
$ns at [expr $now+$time] "plotThroughput $Destino_tcp $archivo"
}
#********* Eventos ****************************************
      $ns at 0.0 "plotVentana      $tcp_F(1) $ventana"
      $ns at 0.0 "plotThroughput   $tcp_D(1) $rendimiento"
      $ns at 6.0 "fin"

# Comenzar la simulación
      $ns run
```

## ANEXO D. Script Variante New Reno en NS-2 TCP Linux (NewReno_Linux.tcl)

```
# Creación objeto simulador
      set ns [new Simulator]
      $ns color 1 blue
      $ns color 2 red
# ************** Creación archivo de traza**************
      set traza1 [open red.nam w]
      $ns namtrace-all $traza1
      set salida [open out.tr w]
#Archivos de Salida
      set ventana [open Ventana w]
      set rendimiento [open Throughput w]
      $ns trace-all $salida
# Procedimiento fin: cierra archivo de traza y se ejecuta NAM
      proc fin { } {
            global ns traza1 salida
            $ns flush-trace
            close $traza1
            close $salida
            exec nam red.nam &
            exit 0
       }
#**********************TOPOLOGÍA*****************************
set Nodos 1
# Creación de Nodos Fuente: F (Emisores)
      for {set i 1} {$i <= $Nodos} {incr i} {
            set F($i) [$ns node]
            $F($i) color red
      }
# Creación de enrutadores y del enlace cuello de botella.
for {set i 1} {$i <= 2} {incr i} {
            set R($i) [$ns node]
            $R($i) shape hexagon
            $R($i) color green
      }
      $ns duplex-link $R(1) $R(2) 0.7Mb 10ms  DropTail
# Creación de Nodos Destino: D (Receptores)
      for {set i 1} {$i <= $Nodos} {incr i} {
            set D($i) [$ns node]
            $D($i) color red
      }
# Creación de enlaces entre Fuentes y el cuello de botella
      for {set i 1} {$i <= $Nodos} {incr i} {
            $ns duplex-link $F($i) $R(1) 10Mb 10ms DropTail
      }
# Creación de enlaces entre el cuello de botella y Destinos
      for {set i 1} {$i <= $Nodos} {incr i} {
            $ns duplex-link $R(2) $D($i) 10Mb 10ms DropTail
      }
```

```
# Tamaño del buffer de la cola del cuello de botella
      $ns queue-limit $R(1) $R(2) 20

# Orientación de los nodos para NAM
      $ns duplex-link-op $F(1) $R(1) orient right-down
      $ns duplex-link-op $R(1) $R(2) orient right
      $ns duplex-link-op $R(2) $D(1) orient right-up

# Etiquetas
      $ns at 0.0 "$R(1) label \"R1\" "
      $ns at 0.0 "$R(2) label \"R2\" "
      $ns at 0.0 "$F(1) label \"F1\" "
      $ns at 0.0 "$D(1) label \"D1\" "

#******** MODULO DE ERROR EN CUELLO DE BOTELLA**************
      set moduloE [new ErrorModel]
      $moduloE set rate_ 0.01
      $moduloE ranvar [new RandomVariable/Uniform]
      $moduloE drop-target [new Agent/Null]
      $ns lossmodel $moduloE $R(1) $R(2)

#*************** AGENTES DE ENVÍO: EMISORES ***************
      for {set i 1} {$i <= $Nodos} {incr i} {
            set tcp_F($i) [new Agent/TCP/Linux]
            $tcp_F($i) set timestamps_ true
            $ns at 0 "$tcp_F($i) select_ca reno"
            $tcp_F($i) set packetSize_ 512
      }
      $tcp_F(1) set fid_ 1

# ********************** AGENTES RECEPTORES*****************
      for {set i 1} {$i <= $Nodos} {incr i} {
            set tcp_D($i) [new Agent/TCPSink]
      }

# ************* CONEXIONES ENTRE EMISOR-RECEPTOR*************
      for {set i 1} {$i <= $Nodos} {incr i} {
            $ns attach-agent $F($i) $tcp_F($i)
            $ns attach-agent $D($i) $tcp_D($i)
            $ns connect $tcp_F($i) $tcp_D($i)
}
#*************AGENTES GENERADORES DE TRÁFICO****************
      for {set i 1} {$i <= $Nodos} {incr i} {
            set ftp($i) [$tcp_F($i) attach-source FTP]
            $ns at 0.0 "$ftp($i) start"
            $ns at 6.0 "$ftp($i) stop"
      }
```

```
#*******************REGISTRO*********************************

proc plotVentana {Fuente_tcp archivo} {
global ns
set time 0.001
set now  [$ns now]
set cwnd [$Fuente_tcp set cwnd_]
puts $archivo "$now $cwnd"
$ns at [expr $now+$time] "plotVentana $Fuente_tcp $archivo"
}

proc plotThroughput {Destino_tcp archivo} {
global ns
set time 0.1
set now    [$ns now]
set nbytes [$Destino_tcp set bytes_]
set throughput [expr ($nbytes * 8.0 / 1000000) / $time]
puts  $archivo  "$now $throughput"
$Destino_tcp set bytes_ 0
$ns at [expr $now+$time] "plotThroughput $Destino_tcp $archivo"
}
#*********************************************************
# Eventos
      $ns at 0.0 "plotVentana       $tcp_F(1) $ventana"
      $ns at 0.0 "plotThroughput    $tcp_D(1) $rendimiento"
      $ns at 6.0 "fin"

# Comenzar la simulación
      $ns run
```

## ANEXO E. Artículo

# SIMULACIÓN DE LAS VARIANTES RENO Y NEW RENO USANDO LA IMPLEMENTACIÓN TCP PROPIA DE NS-2 Y EL MÓDULO TCP LINUX

**Luz Karime Ramírez Aragón**

*Escuela de ingeniería en electrónica, Universidad del Valle*
*Cali, Colombia*

karime.ramirez@correounivalle.edu.co

**Resumen-Este documento presenta un escenario en el que se modela una topología de red de datos con tráfico TCP sometida a condiciones de congestión, mediante un modelo de error en el enlace cuello de botella, se utiliza el lenguaje de programación OTcl (versión orientada a objetos de Tcl (Tool Command Language)) que posteriormente es simulado en el software de simulación Network Simulator 2 (NS-2). En la topología de red se incorporan dos variantes del protocolo TCP (Transmission Control Protocol): Reno y New Reno, con dos implementaciones diferentes: la variante TCP propia del simulador NS-2, denominada variante TCP por defecto en NS-2 y la variante TCP implementada en el módulo NS-2 TCP Linux incorporada en NS-2, denominada variante TCP-Linux. Finalmente se evalúa el desempeño de la red para cada variante en sus dos implementaciones, TCP por defecto y TCP-Linux en NS-2, mediante la observación y el análisis de dos métricas: Ventana de Congestión (cwnd) y Rendimiento (Throughput).**

***Palabras claves:*** NS-2, TCP, Reno, New Reno, Linux TCP

## I. INTRODUCCION

NS-2 es un simulador de eventos discretos (suceso que se genera en un instante de tiempo particular) orientado a objetos, escrito en C++ con un intérprete OTcl (versión orientada a objetos de Tcl: Tool Command Language).

Network Simulator 2 (NS-2) se ha convertido en una de las herramientas de simulación más utilizada en el ámbito académico y de investigación gracias a que se ha distribuido como software libre y de código abierto, lo que ha permitido que estudiantes, ingenieros e investigadores aporten scripts que permiten que esté siempre en proceso de desarrollo [1].

Se realiza la simulación y evaluación del desempeño de dos variantes del protocolo TCP en NS-2 en una red bajo diversas condiciones. En la sección II se desarrolla la metodología implementada para la configuración de la topología de la red. En la sección III se presentan los resultados obtenidos de la simulación de las dos variantes del protocolo TCP en NS-2. En la sección IV se presentan las conclusiones del trabajo.

## II. METODOLOGIA

Se presenta el desarrollo del programa en OTcl donde se muestra la configuración de cada una de las etapas necesarias para la implementación de la topología que modela la red de datos con tráfico TCP sometida a condiciones de congestión.

### *A. Topología*

Con el fin de analizar el comportamiento de las variantes TCP de una forma más detallada se considera una topología de red de 4 nodos como se muestra en la Fig.1.

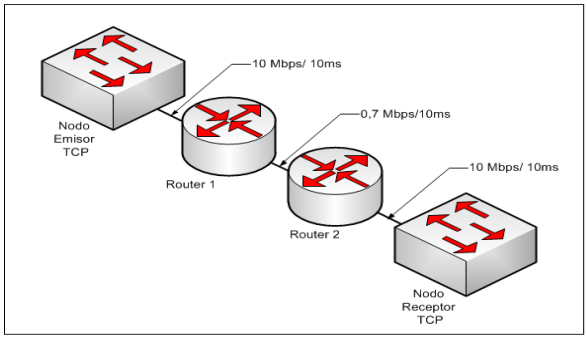


Fig. 1 Modelo de la Topología de red Dumb-bell

Esta topología es el sistema más sencillo para estudiar la congestión ya que el enlace entre los dos routers actúa como cuello de botella (*bottleneck*), el cual debe tener un ancho de banda menor que los otros enlaces para que se produzca congestión. Esta topología se usa comúnmente para estudiar el rendimiento y las variantes del protocolo TCP.

### *B. Nodos*

La creación de los nodos en NS-2 se realiza usando la instancia de procedimiento (instproc): `node` del Simulador cuyo identificador es ns.

### *C. Enlaces*

La creación de un enlace unidireccional o bidireccional en NS-2 se realiza usando el instproc:

- `simplex-link {src dst bw delay qtype}`
- `duplex-link {src dst bw delay qtype}`

De la clase Simulator, donde `src` es un nodo de inicio, `dst` es un nodo destino, `bw` es el ancho de banda del enlace, `delay` es el retardo de propagación del enlace y `qtype` es el tipo de la cola entre el nodo `src` y el nodo `dst`.

### *D. Protocolo y Aplicación*

La red configurada consiste de 4 nodos. En este escenario n0 (F1) transfiere datos a n3 (D1), usando el protocolo de transferencia de archivos FTP sobre el protocolo de transporte TCP/IP. En la Fig. 2 se muestra la topología de la red en el entorno de NAM.

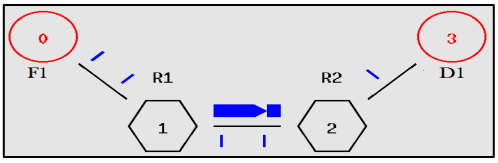


Fig. 2 Topología de red en entorno NAM

La creación del protocolo (agente) TCP/IP en NS-2 consiste en:

- Crear un agente de envío, un agente de recepción y una aplicación usando "`new {…}`".
- Usar el instproc `attach-agent{node agent}` de la clase Simulator para crear una conexión entre cada agente y un nodo de entrada.
- Usar el instproc `connect{src dst}` de la clase Simulator para asociar el agente de envío con el agente de recepción.

En NS-2 se define el emisor TCP (tcp_F1) mediante la clase OTcl `Agent/TCP` y el receptor TCP (tcp_D1) mediante la clase OTcl `Agent/TCPSink`.

La creación del generador de tráfico FTP en NS-2 se realiza usando la clase `Application/FTP` la cual deriva de la clase `Application`.

*E. Modelo de Error*

El modelo de error en NS-2 se realiza usando la clase OTcl ErrorModel. La configuración del modelo de error incluye la configuración de parámetros y de la red. La configuración de parámetros se puede realizar por medio de variables de C++ o de comandos OTcl. La configuración de la red determina donde insertar el modelo de error en un objeto SimpleLink [2].

La configuración del modelo de error en NS-2 consiste en:

- Diseñar y crear una clase de modelo de error en OTcl
- Configurar los parámetros del objeto modelo de error, tales como la probabilidad de error (`rate_`), unidad de error (`unit_`), variable aleatoria (`ranvar_`).

Insertar un modelo de error en la red usando instproc `lossmodel{lossobj from to}` o instproc `link-lossmodel{lossobj from to}` de la clase Simulator.

En la Fig. 3 se presenta la configuración del Modelo de Error implementado en el enlace cuello de botella.

```
#********* MODULO DE ERROR EN CUELLO DE BOTELLA**************
41   set moduloE [new ErrorModel]
42   $moduloE set rate_ 0.01
43   $moduloE ranvar [new RandomVariable/Uniform]
44   $moduloE drop-target [new Agent/Null]
45   $ns lossmodel $moduloE $R(1) $R(2)
```

Fig. 3 Configuración del modelo de error en el cuello de botella

*F. Métricas*

Las métricas a observar y analizar en la red diseñada para evaluar el desempeño de la red son la Ventana de Congestión (*cwnd*) y el Rendimiento (*Throughput*).

En la Fig. 4 se presenta la configuración del procedimiento que permite obtener la ventana de congestión durante la transmisión. El procedimiento llamado plotVentana se encarga de registrar cada 0.001s (variable time) el valor de la ventana de congestión en paquetes (`cwnd_`) en el archivo de salida "Ventana".

```
#Archivos de Salida para guardar cwnd
46   set ventana [open Ventana w]
#Procedimiento para registrar el valor de ventana de congestión
47   proc plotVentana {Fuente_tcp archivo} {
48   global ns
49   set time 0.001
50   set now [$ns now]
51   set cwnd [$Fuente_tcp set cwnd_]
52   puts $archivo "$now $cwnd"
53   $ns at [expr $now+$time] "plotVentana $Fuente_tcp $archivo"
54   }
#Llamada del Procedimiento.

55   $ns at 0.0 "plotVentana  $tcp_F(1) $ventana"
```

Fig. 4 Registro ventana de congestión (cwnd)

En la Fig. 5 se presenta la configuración del procedimiento que permite determinar el rendimiento. El procedimiento llamado plotThroughput se encarga de registrar cada 0.1s el valor de la cantidad de bytes recibidos por el receptor (bytes_) en el archivo de salida "Throughput". Se calcula el valor del rendimiento en Mbps.

```
#Archivos de Salida para guardar el Throughput
56   set rendimiento [open Throughput w]
#Procedimiento para registrar el Throughput
57   proc plot Throughput {Destino_tcp archivo} {
58   global ns
59   set time 0.1
60   set now  [$ns now]
61   set nbytes [$Destino_tcp set bytes_]
62   set throughput [expr ($nbytes * 8.0 / 1000000) / $time]
63   puts  $archivo  "$now $throughput"
64   $Destino_tcp set bytes_ 0
65   $ns at [expr $now+$time] "plotThroughput $Destino_tcp
$archivo"
66   }
#Llamada del Procedimiento.
67   $ns at 0.0 " plotThroughput  $tcp_D(1) $rendimiento"
```

Fig. 5 Registro del Throughput

El tiempo de la simulación se obtiene mediante el insproc now{} de la clase Simulator. Ambos procedimientos son recursivos, después de cada intervalo de tiempo se llama de nuevo.

En la Tabla I se presentan algunas de las variables utilizadas en el script de las clases Agent/TCP y Agent/TCPSink.

TABLA I
Variables En NS-2

| Variable OTcl | Valor por defecto | Descripción |
|---|---|---|
| window_ | 20 | Límite superior del tamaño de la ventana |
| numdupacks_ | 3 | Número de ACKs duplicados el cual activa Fast Retransmit |
| packetSize_ | 512 | Tamaño del paquete TCP en bytes |
| cwnd_ | 0 | Tamaño de la ventana de congestión en paquetes |
| ssthresh_ | 0 | Umbral de arranque lento |
| bytes_ | 0 | Número de bytes recibidos |
| ndatapack_ | 0 | Número de paquetes transmitidos por la conexión |

*G. Archivos*

Para el análisis de los resultados se deben filtrar los archivos de salida obtenidos de la simulación que se presentan en la Fig. 6

```
set salida      [open out.tr w]
set ventana     [open Ventana w]
set rendimiento [open Throughput w]
```

Fig. 6 Archivos de Salida

El archivo out.tr es el generado por el simulador que contiene los eventos que ocurren durante la simulación de la red. El archivo Ventana es el generado en el procedimiento para el registro de la ventana de congestión, donde se tiene el valor de cada instante de cwnd_. El archivo Throughput es el generado en el procedimiento para el registro de los bytes recibidos en el nodo destino a partir del cual se calcula el rendimiento de la red.

El filtro del archivo out.tr se obtiene mediante el programa AWK de la Fig. 7 con el cual se determinan la cantidad de paquetes transmitidos, perdidos y recibos durante la transmisión.

```
BEGIN {
        recibidos = 0;          #paquetes recibidos (r)
        transmitido = 0;        #paquetes transmitidos (-)
        perdido = 0;            #paquetes perdidos (d)
}

{
        evento          = $1;   #Columna del archivo out.tr
        from_Nodo       = $3;
        to_Nodo         = $4;
        size            = $6;
        fuente          = $9;
        destino         = $10;

        if (evento == "r" && from_Nodo==2 && to_Nodo==3 && size == 552 && fuente==0.0 && destino==3.0)
        recibidos++;
        if (from_Nodo==0 && to_Nodo==1 && evento == "-"  && size == 552)
        transmitido++;
        if (evento == "d")
        perdido++;
}
END {
        printf("Paquetes transmitidos: %d\n", transmitido);
        printf("Paquetes recibidos: %d\n", recibidos);
        printf("Paquetes perdidos: %d\n", perdido);
}
```

Fig. 7 Script AWK

*H. Pruebas*

1) Ejecución del script en OTcl y verificación que el programa esté libre de errores.

2) realizar la configuración de la variante TCP en NS-2 y obtener los archivos de salida de la ventana de congestión y del rendimiento.

- variante Reno ejecutar el script "Reno_NS2.tcl" donde se configura el agente Agent/TCP/Reno.
- variante New Reno ejecutar el script "NewReno_NS2.tcl" donde se configura el agente Agent/TCP/Newreno.

3) Realizar la configuración de la variante TCP en el módulo de NS-2 Linux TCP y obtener los archivos de salida "Ventana" y "Throughput".

- Variante Reno ejecutar el script "Reno_Linux.tcl" donde se configura el agente Agent/TCP/Linux y el nombre para la variante Reno: naive_reno.
- Variante New Reno ejecutar el script "NewReno_Linux.tcl" donde se configura el agente Agent/TCP/Linux y el nombre para la variante New Reno: reno.

4) Determinar los paquetes transmitidos a partir del archivo out.tr de cada una de las simulaciones mediante el siguiente filtro en awk:

```
>> awk -f '$1=="-" && $3==0 && $4==1
{print $2,23}' out.tr>transmitidos
```

El filtro consiste en comparar las columnas evento, nodo de entrada y nodo de salida del archivo out.tr para determinar el tiempo en que se realiza la transmisión de un paquete. Con la sentencia print se registra el tiempo de transmisión en un archivo llamado transmitidos.

5) Determinar los paquetes perdidos a partir del archivo out.tr mediante el siguiente filtro en awk:

```
>> awk -f '/d/ {print $2,23}' out.tr
> perdidos
```

El filtro consiste en registrar en un archivo llamado perdidos los tiempos en los que ocurre una pérdida de paquete.

6) Para cada variante se realizan 3 réplicas diferentes del experimento introduciendo aleatoriedad en la simulación mediante un generador de números aleatorios (*Random number generator (RNG)*).

Un RNG se inicializa con una semilla (seed). La semilla identifica la ubicación inicial en una secuencia-psudo aleatoria, donde un RNG inicia la selección de números [2].

## III RESULTADOS Y DISCUSION

Se presentan las gráficas de los resultados obtenidos para la ventana de congestión (*cwnd*) y el rendimiento (Throughput) de cada variante de TCP para una de las réplicas realizadas. Los datos fueron tomados de los archivos de traza obtenidos por la simulación de la topología en NS-2; mediante el uso de AWK se realizan filtros sobre los archivos out.tr para determinar los paquetes transmitidos, paquetes enviados, paquetes perdidos, tiempos de retransmisión de los paquetes, entre otros.

### *A. Variante TCP Reno*

*1) TCP Reno por defecto***.** En la Fig. 8, la línea de color azul representa el valor de la ventana de congestión (cwnd) en el tiempo. En la parte superior los puntos de color violeta representan los tiempos de transmisión de los paquetes. La línea discontinua de color negro representa los instantes de pérdida de paquetes. La línea continua de color rojo representa los instantes de retransmisión.

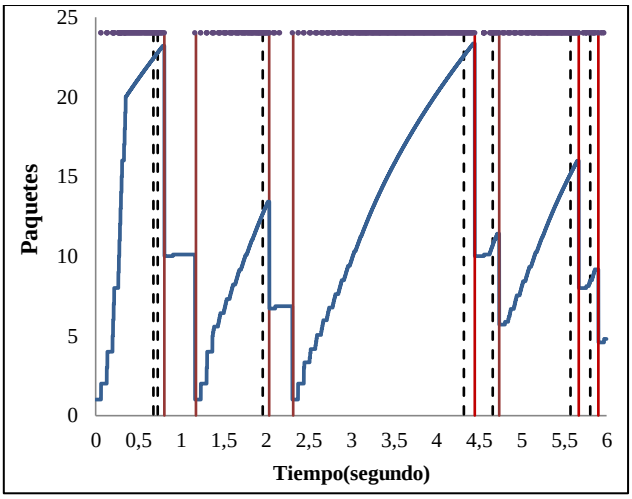


Fig. 8 Comportamiento Control de Congestión TCP Reno

Al inicio de la conexión se puede observar un crecimiento rápido de *cwnd* que corresponde a la ejecución del algoritmo de Arranque Lento (*Slow Start*) y mediante los puntos que representan los paquetes transmitidos se puede observar que la velocidad de transmisión del emisor aumenta. La fase de Slow Start continúa hasta que se presenta congestión en la red, determinada por la pérdida de paquetes. En el instante 0.6767s se pierde el paquete con número de secuencia 88. En el instante 0.8050s se presenta la retransmisión del paquete perdido, observando que el valor de *cwnd* se reduce a la mitad del valor de la ventana (window=20) estableciendo el valor del umbral de Slow Start (*ssthresh*=10). La retransmisión del paquete 88 perdido se ejecuta después de que el emisor ha recibido tres ACKs duplicados del

paquete con número de secuencia 87, entrando en la fase del algoritmo Retransmisión Rápida / Recuperación Rápida (*Fast Retransmit / Fast Recovery*).

La fase de *Fast Recovery* termina al retransmitir el paquete perdido. En el instante 1.1764s ocurre un *Timeout*, es decir que el temporizador de retransmisión ha expirado, de esta manera *cwnd* se reestablece en 1, iniciando de nuevo el algoritmo de *Slow Start* hasta alcanzar el valor de *ssthresh*. Cuando *cwnd* supera el valor de *ssthresh* se cambia al algoritmo de Prevención de la Congestión (Congestion Avoidance) aumentando *cwnd* de manera lineal, hasta que se detectan pérdidas de datos adicionales. Después de estos dos eventos de pérdida se puede observar que se repite el mismo comportamiento.

En la Tabla II se presenta el seguimiento de los paquetes perdidos en la conexión con sus tiempos de retransmisión y el valor respectivo de la ventana de congestión.

TABLA II
Retransmisión de Paquetes en TCP Reno

| Num.Seq Paquete Perdido | Tiempo de Pérdida | Flag | Tiempo de Retransmi-sión | cwnd | Reenvío después del: | Retransmisión por |
|---|---|---|---|---|---|---|
| 88 | 0,6767 | A | 0,805 | 10 | 3ACK duplicado | *Fast Retransmit* |
| 96 | 0,7272 | A | 1,1764 | 1 | | *Timeout* |
| 188 | 1,9608 | A | 2,0387 | 6,7081 | 3ACK duplicado | *Fast Retransmit* |
| 201 | 2,0365 | A | 2,3185 | 1 | | *Timeout* |
| 477 | 4,3234 | A | 4,4517 | 10 | 3ACK duplicado | *Fast Retransmit* |
| 511 | 4,6624 | A | 4,7386 | 5,7038 | 3ACK duplicado | *Fast Retransmit* |
| 634 | 5,5763 | A | 5,6731 | 7,9933 | 3ACK duplicado | *Fast Retransmit* |
| 660 | 5,808 | A | 5,9031 | 4,5834 | 3ACK duplicado | *Fast Retransmit* |

*2) TCP Linux Reno*. En la Fig. 9 se utiliza la misma nomenclatura de la Fig. 8 para la ventana de congestión, tiempo de transmisión de paquetes, tiempo de perdida de paquetes y retrasmisión de paquetes

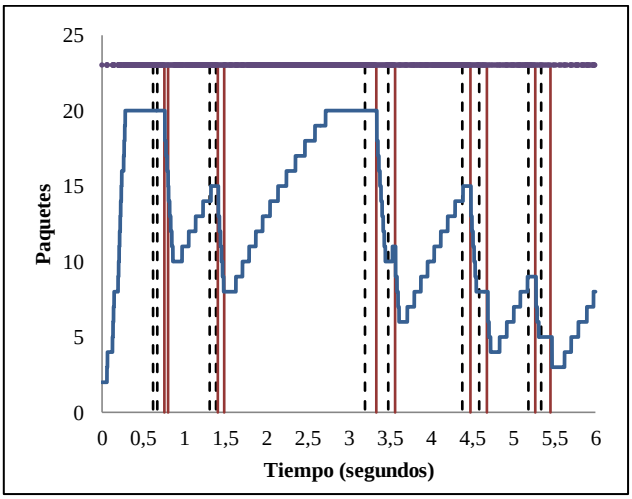


Fig. 9 Comportamiento Control de Congestión TCP Linux Reno

La Fig. 9 muestra el comportamiento de la ventana de congestión (*cwnd*) simulando la versión TCP Linux Reno. La ejecución del algoritmo *Slow Start* se observa solo al inicio de la conexión. Cuando se detecta la pérdida del paquete 88 por *Fast Retransmit* se inicia la fase del algoritmo *Fast Retransmit / Fast Recovery*, el valor de *cwnd* se reduce gradualmente para evitar más congestión hasta el umbral de arranque lento (*ssthresh*) que se establece en la mitad del valor de *cwnd* que existía antes de la pérdida. Después de que los dos paquetes perdidos son retransmitidos, se inicia la fase del algoritmo *Congestion Avoidance*, aumentando linealmente la ventana de congestión.

Se evidencia una de las modificaciones y mejoras de Linux TCP en cuanto al algoritmo de *Fast Retransmit/ Fast Recovery* con SACK que ayuda a acelerar la detección de paquetes perdidos, ya que con SACK los ACKs contienen información adicional que hace que el emisor no tenga que esperar un RTT para retransmitir el paquete 96 después de la retransmisión del paquete 88. De esta manera se observa que los Timeouts con la versión Reno por defecto en NS-2 han desaparecido.

En la Tabla III se presenta el seguimiento de los paquetes perdidos en la conexión con sus tiempos de retransmisión y el valor respectivo de la ventana de congestión.

TABLA III
Retransmisión de Paquetes en TCP Linux Reno

| Num.Seq Paquete Perdido | Tiempo de Pérdida | Flag | Tiempo de Retransmisión | cwnd | Reenvío después de | Retransmisión por |
|---|---|---|---|---|---|---|
| 88 | 0,6219 | A | 0,7592 | 20 | 4ACK duplicado | *Fast Retransmit* |
| 96 | 0,6732 | ---- | 0,8043 | 16 | 12ACK duplicado | *Fast Retransmit* |
| 188 | 1,3101 | A | 1,409 | 15 | 4ACK duplicado | *Fast Retransmit* |
| 201 | 1,3872 | ---- | 1,4861 | 8 | ACK duplicado | *Fast Retransmit* |
| 477 | 3,1974 | A | 3,3348 | 20 | 4ACK duplicado | *Fast Retransmit* |
| 511 | 3,4801 | A | 3,5632 | 11 | 4ACK duplicado | *Fast Retransmit* |
| 634 | 4,3799 | A | 4,4787 | 15 | 4ACK duplicado | *Fast Retransmit* |
| 660 | 4,5854 | A | 4,6787 | 8 | 4ACK duplicado | *Fast Retransmit* |
| 713 | 5,1834 | A | 5,2666 | 9 | 4ACK duplicado | *Fast Retransmit* |
| 728 | 5,3385 | A | 5,4515 | 5 | 4ACK duplicado | *Fast Retransmit* |

En la Fig. 10 se presenta el rendimiento (*Throughput*) de ambas implementaciones de la variante Reno. Se puede observar que:

- El *Throughput* aumenta a medida que la carga en la red es mayor en ambas implementaciones.
- Cuando el *throughput* llega a un valor de cero es porque existe un Timeout. En la Fig. 8 se puede observar que en ese momento no hay transmisión de paquetes mediante los puntos de la parte superior que indican los tiempos de los paquetes transmitidos.
- El rendimiento de la red en la implementación de la variante TCP Linux Reno es mayor.

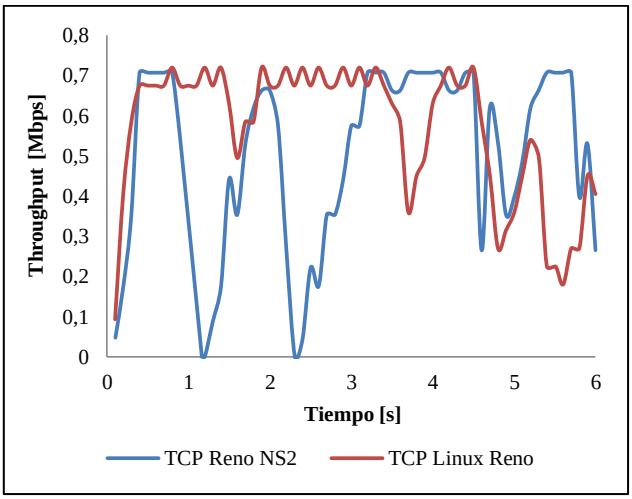


Fig. 10 Throughput Variante TCP Reno

*B. Variante TCP New Reno*

1) TCP New Reno por defecto. En la Fig. 11: la línea de color azul representa el valor de la ventana de congestión (cwnd) en el tiempo. En la parte superior los puntos de color violeta representan los tiempos de transmisión de los paquetes. La línea discontinua de color negro representa los instantes de pérdida de paquetes. La línea continua de color rojo representa los instantes de retransmisión.

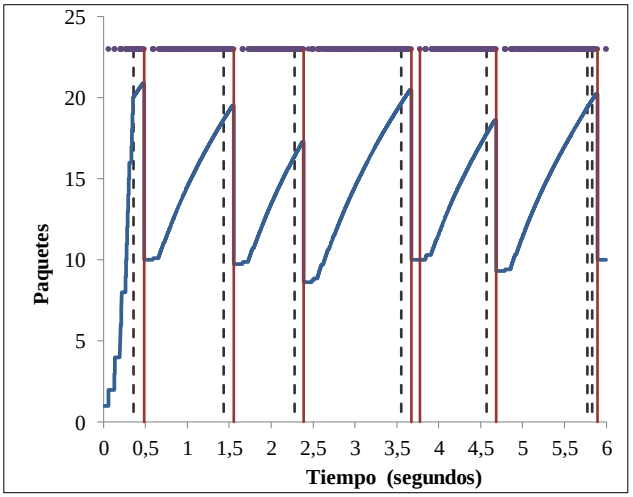


Fig. 11 Comportamiento Control de Congestión TCP New Reno

En la Fig. 11 se muestra el comportamiento de la ventana de congestión (*cwnd*) simulando la versión TCP New Reno. La ejecución del algoritmo *Slow Start* se observa solo al inicio de la conexión. La fase de *Slow Start* continúa hasta que se presenta congestión en la red determinada por la pérdida de paquetes. En el instante 0.4833s se presenta la retransmisión del paquete perdido, observando que el valor de *cwnd* se reduce a la mitad y se establece el valor de *ssthresh*.

La retransmisión del paquete 37 perdido se ejecuta después de que el emisor ha recibido tres ACKs duplicados del paquete con número de secuencia 36, entrando en la fase de *Fast Retransmit / Fast Recovery*.

Se observa que durante el estado *Fast Recovery* la ventana de congestión se mantiene en el mismo valor (*cwnd*=10) hasta que en el instante 0.5905s se recibe en el emisor el ACK del paquete 56, que representa el Punto de Recuperación de la fase *Fast Retransmit / Fast Recovery*; por tanto termina la fase de *Fast Recovery* y empieza la fase del algoritmo *Congestion Avoidance*, evidenciando el aumento lineal de *cwnd* hasta una nueva retransmisión.

La Tabla IV registra los tiempos de retransmisión de cada paquete perdido con el valor de *cwnd*. Se registra el paquete con el número de secuencia más alto enviado después del paquete perdido, denominado Punto de Recuperación.

TABLA IV
Retransmisión de Paquetes en TCP New Reno

| Num.Seq Paquete Perdido | Tiempo de Pérdi-da | Flag | Tiempo de Retransm isión | cwnd | Reenví o después del: | Retransmi -sión por | Punto de Recupera-ción |
|---|---|---|---|---|---|---|---|
| 37 | 0,3613 | A | 0,4833 | 10 | 3ACK duplica do | *Fast Retransmit* | Paquete 56 |
| 196 | 1,4367 | A | 1,5524 | 9,755 | 3ACK duplica do | *Fast Retransmit* | Paquete 215 |
| 316 | 2,2833 | A | 2,3864 | 8,627 | 3ACK duplica do | *Fast Retransmit* | Paquete 334 |
| 506 | 3,5509 | A | 3,6729 | 10 | 3ACK duplica do | *Fast Retransmit* | Paquete 525 |
| 512 | 3,5887 | ---- | 3,7739 | 10 | ACK del paquete 511 | *ACK Parcial* | |
| 648 | 4,5741 | A | 4,6836 | 9,308 | 3ACK duplica do | *Fast Retransmit* | Paquete 667 |
| 828 | 5,7740 | A | 5,8961 | 10 | 3ACK duplica do | *Fast Retransmit* | Paquete 847 |
| 837 | 5,8308 | ---- | El paquete no se recupera. | | | | |

7.1.1 2) TCP Linux New Reno. En la Fig. 12 se utiliza la misma nomenclatura de la Fig. 11 para la ventana de congestión, tiempo de transmisión de paquetes, tiempo de perdida de paquetes y retrasmisión de paquetes.

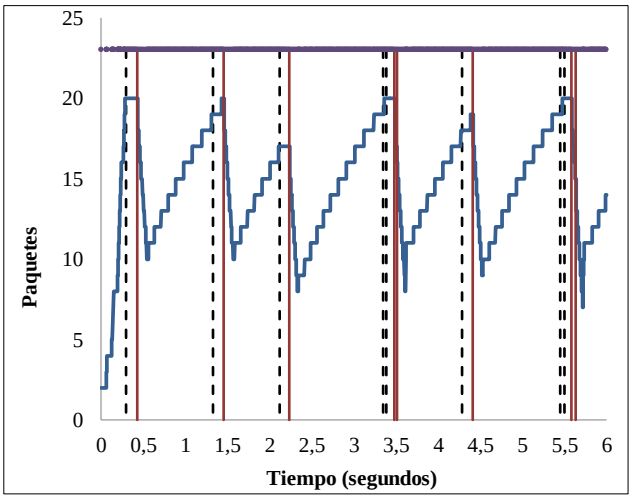


Fig. 12 Comportamiento Control de Congestión TCP Linux New Reno

En la Fig. 12 cuando el emisor recibe ACKs de paquetes nuevos, es decir que no recibe un ACK parcial o menor al punto de recuperación se salen de la fase *Fast Recovery* y se observa el aumento lineal de *cwnd* gracias al algoritmo *Congestion Avoidance*.

Igual que en la implementación anterior (New Reno por defecto en NS-2) se observa que la retransmisión de varios paquetes perdidos se puede realizar sin observar un restablecimiento de *cwnd*. Sin embargo, se observa que la segunda retransmisión (t=3.5138s) se realiza después de 0.0323s de la primera retransmisión, es decir en un tiempo más corto que en la implementación de New Reno anterior (0.1009s).

En la Tabla V se presenta el seguimiento de los paquetes perdidos en la conexión con sus tiempos de retransmisión y el valor respectivo de la ventana de congestión, donde se observa que la retransmisión de los paquetes perdidos ya no necesariamente se realiza después del tercer ACK duplicado sino del cuarto.

En la Fig. 13 se presenta el rendimiento (*Throughput*) de ambas implementaciones de la variante New Reno. Se puede observar que:

- El *Throughput* aumenta a medida que la carga en la red es mayor en ambas implementaciones.
- El *throughput* no llega a un valor de cero porque los eventos de Timeout han desaparecido por la mejora del algoritmo de Fast Recovery.
- El rendimiento de la red en la implementación de la variante TCP Linux New Reno es mayor.

TABLA V
Retransmisión de Paquetes en TCP Linux New Reno

| Num. Seq Paquete Perdido | Tiempo de Pérdida | Flag | Tiempo de Retransmisión | cwnd | Reenvío | Retransmisión por | Punto de Recuperación |
|---|---|---|---|---|---|---|---|
| 37 | 0,2947 | A | 0,4317 | 20 | 4ACK duplicado | *Fast Retransmit* | Paquete 59 |
| 196 | 1,3257 | A | 1,4566 | 20 | 4ACK duplicado | *Fast Retransmit* | Paquete 218 |
| 316 | 2,1194 | A | 2,2311 | 17 | 4ACK duplicado | *Fast Retransmit* | Paquete 335 |
| 506 | 3,3506 | A | 3,4815 | 20 | 3ACK duplicado | *Fast Retransmit* | Paquete 528 |
| 512 | 3,3827 | | 3,5138 | 17 | 9ACK duplicado | *ACK Parcial* | |
| 648 | 4,2855 | A | 4,4100 | 19 | 4ACK duplicado | *Fast Retransmit* | Paquete 669 |
| 828 | 5,4489 | A | 5,5798 | 20 | 4ACK duplicado | *Fast Retransmit* | Paquete 850 |
| 837 | 5,5002 | | 5,6313 | 15 | 12ACK duplicado | *ACK Parcial* | |

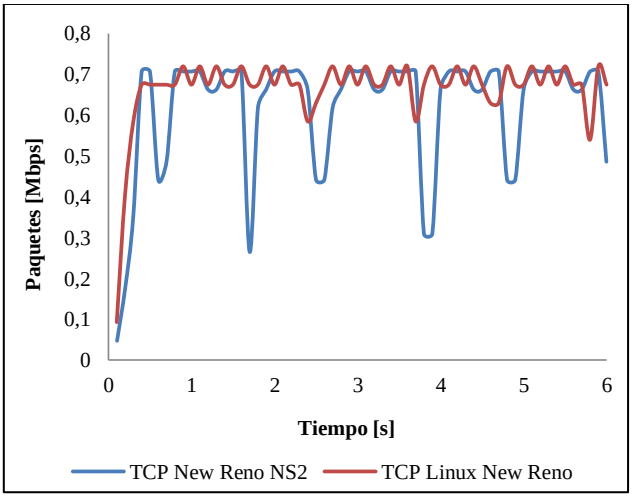


Fig. 13 Throughput Variante TCP New Reno.

## IV CONCLUSIONES

Network Simulator 2 como herramienta de simulación permite al usuario implementar escenarios para modelar redes de comunicación con diferentes protocolos, aplicaciones, enlaces y elementos de red. Gracias a su distribución libre ha permitido el desarrollo de nuevos protocolos y algoritmos que representan mejoras en el rendimiento de la red. La gran cantidad de scripts de código abierto que se encuentran en la literatura y en la web pueden ser libremente accedidos y modificados, permitiendo al usuario tener un proceso de aprendizaje y manejo de la herramienta con mayor facilidad.

El desarrollo del módulo de simulación de la red de datos en NS-2 se realiza mediante un script en lenguaje OTcl incorporando los elementos de red, enlaces, protocolos y aplicaciones de una topología sencilla sometida a congestión que permite evaluar el desempeño de los algoritmos de control de congestión de las variantes Reno y New Reno del protocolo de TCP.

El modelo de la red de datos con tráfico TCP se puede someter a condiciones de congestión mediante el cambio de la capacidad del buffer o mediante la inclusión de un Modelo de Error con una probabilidad de error en la trasmisión de paquetes.

Las características adicionales de los algoritmos de control de congestión implementados en Linux-TCP permiten tener mejoras en el rendimiento de la red y la detección de la congestión se realiza de una forma más rápida y precisa. Mediante la ventana de congestión (*cwnd*) se puede observar que en las implementaciones de la variante TCP en el módulo NS-2 TCP Linux, las retransmisiones de los paquetes perdidos se realizan más rápido y por tanto el rendimiento (*throughput*) de la red es mayor.

Los algoritmos de control de congestión Arranque Lento (*Slow Start*), Prevención de Congestión (*Congestion Avoidance*), Retransmisión Rápida (*Fast Retransmit*) y Recuperación Rápida (*Fast Recovery*) presentan el comportamiento básico del control de congestión AIMD (*Additive Increase / Multiplicative Decrease)* tanto para la implementación de la variante TCP en NS-2 y en la implementación en el módulo NS-2 TCP Linux.

## RECONOCIMIENTOS

Extiendo mi reconocimiento a Oscar Polanco Sarmiento quien me dirigió durante la ejecución de los experimentos.

## REFERENCIA

[1] E. ALTMAN y T. JIMENEZ, *NS Simulator for Beginners, Morgan & Claypool*, 2012, p. 184.

[2] T. ISSARIYAKUL y E. HOSSAIN, *Introduction to Network Simulator NS2*, Springer, 2009, p. 438.